



COMSATS University Islamabad, Lahore Campus

Block-B, Department of Electrical and Computer Engineering

1.5 KM Defence Road, Off Raiwind Road, Lahore

CPE344 –Digital System Design

Lab Manual for Fall 2024

Lab Resource Person

Muhammad Usman

Theory Resource Person

Ms. Wajeeha Khan

Supervised By

Ms. Wajeeha Khan

Name: _____ Registration Number: CUI/ - - /LHR

Program: _____ Batch: _____

Semester _____

Revision History

S. No.	Update	Date	Performed by
1	Lab Manual Preparation	Jan-2015	Nauman Hafeez
2	Lab Manual Review	Jan-2015	Dr. Umar Farooq
3	Layout modifications	Feb-2015	Nauman Hafeez
4	Lab Manual Review	August-2015	Engr. Irfan Ali
5	Lab Manual Review /Modification	August-2016	Engr. Irfan Ali
6	Lab Manual Preparation / Update / Modification	Feb-2018	Engr. Irfan Ali
7	Lab Manual Modification	Sept- 2018	Engr. Wajeeha Khan
8	Lab Manual Modification	Feb 2020	Engr. Wajeeha Khan
9	Lab Manual Modification	Sept 2021	Engr. Wajeeha Khan
10	Lab Manual Modification	Feb 2022	Engr. Muhammad Usman

Preface

The lab is meant for design and evaluation of control and data structures for digital systems. Hardware design language is used to describe and design both behavioral and Register Transfer Level (RTL) architectures and control units with a microprogramming emphasis. This lab manual explains how to go about designing complex, high-speed digital systems. The use of modern EDA tools in the design, simulation, synthesis and implementation is explored. Application of a hardware description language such as Verilog or VHDL to model digital systems at Behavior and RTL level is studied. Field programmable gate arrays (FPGA) are used in the laboratory exercises as a vehicle to understand complete design-flow of an integrated circuit. Advanced methods of logic minimization and state-machine design are discussed. Design and implementation of digital system building blocks such as arithmetic circuits, processors, I/O modules, communication system blocks, frequency generators, memories etc. is included. Laboratories and projects are an integral part of this course that culminates in a comprehensive design exercise.

Books

1. **Book1:** Digital Design by Frank Vahid
2. **Book2:** Digital Systems Design using VHDL by Charles H. Roth, Jr.
3. **Book3:** Principles of Digital Systems Design by Charles H. Roth and L. K. John

Reference Books

1. Digital Electronics and Design with VHDL by Volnei A. Pedroni
2. Circuit Design and Simulation with VHDL by Volnei A. Pedroni Reference Books for Manual

Learning Outcomes

After successful completion of this module, you will be able to:

Theory CLOs:

1. Analyze and design the working of advanced combinational and sequential logic-based systems using the classical principals of digital logic design. (PLO3-C5)
2. Design of digital systems in a hierarchical and top-down manner using register-transfer logic (RTL) approach. (PLO3-C5)

Lab CLOs:

3. To design the digital systems based on HDL modelling techniques using VHDL. (PLO3-C5)
4. Reproduce the response of the designed digital systems using the software tool and hardware platform (PLO5-P3)
5. To explain and write effective project reports for the complex Engineering problem (PLO10- A3)

CLO – PLO Mapping

PLO CLOs	PLO1	PLO2	PLO3	PLO4	PLO5	PLO6	PLO7	PLO8	PLO9	PLO10	PLO11	PLO12	Cognitive level	Psychomo tor Level	Affective Level
CLO1			x										C5		
CLO2			x										C5		
CLO3			x										C5		
CLO4					x									P3	
CLO5										x					A3

CLOs – Lab Experiments Mapping

Lab Exp CLOs	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6	Exp 7	Exp 8	Exp 9	Exp 10	Exp 11	Exp 12	Exp 13	Exp 14
Lab CLO 2	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3

Grading Policy

Lab Assignments: i. Lab Assignment 1 Marks (Lab marks from Labs 1-3) ii. Lab Assignment 2 Marks (Lab marks from Labs 4-6) iii. Lab Assignment 3 Marks (Lab marks from Labs 7-9) iv. Lab Assignment 4 Marks (Lab marks from Labs 10-12)	25%
Lab Mid Term = $0.5 * (\text{Lab Mid Term exam}) + 0.5 * (\text{average of lab evaluation of Lab 1-6})$	25%
Lab Terminal = $0.5 * (\text{Complex Engineering Problem}) + 0.375 * (\text{average of lab evaluation of Lab 7-12}) + 0.125 * (\text{average of lab evaluation of Lab 1-6})$	50%

Total (lab)

100%

List of equipment

Altera Cyclone IV E -DE2-115 FPGA
Altera Cyclone SoC V SE -DE1 FPGA

Software Resources

Quartus 13.1

Lab Instructions

1. This lab activity comprises of three parts: Pre-lab, Lab Exercises, and Post-Lab Viva session.
2. The students should perform and demonstrate each lab task separately for step-wise evaluation (please ensure that course instructor/lab engineer has signed each step after ascertaining its functional verification)
3. Only those tasks that completed during the allocated lab time will be credited to the students. Students are however encouraged to practice on their own in spare time for enhancing their skills.

Complex Engineering Problem Report Instructions

All questions should be answered precisely to get maximum credit. Lab report must ensure following items:

1. Introduction
2. Literature Review
3. Problem Implementation
4. Results
5. Critical Analysis/ Conclusion
6. Bibliography
7. Appendix

Laboratory Regulations and Safety Instructions

1. It is the duty of all concerned who use any electrical laboratory to take all reasonable steps to safeguard
the health and safety of themselves and all other users and visitors.
2. Be s-ure that all equipment is properly working before using them for laboratory exercises. Any
defective equipment must be reported immediately to the Lab. Instructors or Lab. Technical Staff.
3. Students are allowed to use only the equipment provided in the experiment manual.
4. Power supply terminals connected to any circuit are only energized with the presence of the Instructor
or Lab. Staff.
5. Avoid any part of your body to be connected to the energized circuit and ground.
6. Switch off the equipment and disconnect the power supplies from the circuit before leaving the
laboratory.
7. Observe cleanliness and proper laboratory housekeeping of equipment and other related accessories.
8. Equipment should not be removed, transferred to any location without permission from the laboratory
staff.
09. Students are not allowed to use any equipment without proper orientation.
10. Do not eat food, drink beverages, or chew gum in the laboratory.

Table of Contents

Revision History	2
Preface	3
Books	4
Reference Books	Error! Bookmark not defined.
Reference Books for Manual	4
Learning Outcomes	4
CLO – PLO Mapping	4
CLOs – Lab Experiments Mapping	5
Grading Policy	5
List of equipment	6
Software Resources	6
Lab Instructions	6
Lab Report Instructions	6
Laboratory Regulations and Safety Instructions	7
Table of Contents	8
LAB # 1: Introduction to VHDL and Altera Quartus with the design of Full Adder and reproduce the output of FPGA board	11
<i>Objectives</i>	<i>11</i>
<i>Part 1 -Familiarize yourself with VHDL and Quartus:</i>	<i>11</i>
<i>Part 2 –Design and Implementation of Four-bit adder:</i>	<i>14</i>
LAB # 2: To design the combinational and sequential circuit components using VHDL and reproduce the output on the FPGA board	24
<i>Objectives</i>	<i>24</i>
<i>Part 1 –Design and Implementation of Combinational Logic</i>	<i>24</i>
<i>Part 2 –Design and Implementation of Sequential Logic</i>	<i>27</i>
LAB # 3: To design the finite state machines using VHDL and reproduce the results on FPGA board	34
<i>Objectives</i>	<i>34</i>
<i>Part 1 – Introduction, specification, design and testing of Mealy FSM</i>	<i>34</i>
<i>Part 2 – Introduction, specification, design and testing of Moore FSM</i>	<i>37</i>

LAB # 4: To design the counter and sequence detector using VHDL and reproduce the sequence on the FPGA board	40
<i>Objectives</i>	40
<i>Part 1 – Introduction, specification, design and testing of counter</i>	40
<i>Part 2 – Introduction, specification, design and testing of Sequence Detector</i>	41
Lab 5 (a) To design the complex counter and using VHDL and reproduce the results on the FPGA board	43
Lab 5 (b) To design the vending machine using VHDL and reproduce the results on the FPGA(open ended)	43
board	Error! Bookmark not defined.
<i>Objectives</i>	43
<i>Part 1 – Introduction, specification, design and testing of a Complex Counter</i>	43
<i>Part 2 Lab 5 (b) – Introduction, specification, design and testing of Vending Machine</i>	44
<i>Machine</i>	44
LAB # 6: To design the array and binary multiplier using VHDL and reproduce the output on the FPGA board	48
<i>Objectives</i>	48
<i>Part 1 – Introduction, specification, design and testing of an array multiplier</i>	48
<i>Part 2 – Introduction, specification, design and testing of binary multiplier (Shift and Add Multiplier)</i>	50
LAB # 7: To design the signed binary multiplier using Booth’s algorithm in VHDL and reproduce the results on FPGA board	54
<i>Objectives</i>	54
<i>Part 1 – Introduction, specification, design and testing of a signed binary number multiplier</i>	54
-LAB # 8: To design the binary and signed binary divider using VHDL and reproduce output on FPGA board	58
<i>Objectives</i>	58
<i>Part 1 – Introduction, specification, design and testing of a binary divider</i>	58
<i>Part 2 – Introduction, specification, design and testing of a signed binary number divider</i>	60
LAB # 9: To design the traffic light controller using VHDL and reproduce the output on the FPGA board	64
<i>Objectives</i>	64
<i>Part 1 - Introduction, specification, design and testing of two way traffic light controller</i>	64
LAB # 10: To design the memories (RAM and ROM) using VHDL and reproduce output on FPGA board	68
<i>Objectives</i>	68
<i>Part 1 – Introduction, specification, design and testing of RAM</i>	68
<i>Part 2 – Introduction, specification, design and testing of ROM</i>	69
LAB # 11: To design the keypad scanner using VHDL and reproduce the output on the FPGA board	73
<i>Objectives</i>	73

<i>Part-1 Introduction, specification, design and testing of a Complex Counter</i>	73
<i>Part- 2 FSM of controller and decoder design of keypad scanner</i>	75
LAB # 12: To design the finite impulse response (FIR) filter using VHDL and reproduce output on vector waveform	77
<i>Objective</i>	77
<i>Part 1 – Introduction, specification, design and testing of direct form FIR filter</i>	77
<i>Part 2 – Introduction, specification, design and testing of time multiplexed architecture of FIR filter</i>	78
LAB # 13: To design the fast fourier transform (FFT) processor using VHDL and reproduce the output on vector waveform	Error! Bookmark not defined.
<i>Objective</i>	Error! Bookmark not defined.
<i>Part- 1 Introduction, specification, design and testing of FFT Processor</i>	Error! Bookmark not defined.
<i>Part- 2 Design the controller of 16-Point FFT Processor</i>	Error! Bookmark not defined.
LAB # 14: To design the video graphics array (VGA) controller using VHDL and reproduce output on vector waveform	Error! Bookmark not defined.
<i>Objectives</i>	Error! Bookmark not defined.
<i>Part- 1 Introduction, specification, design and testing of VGA Controller</i>	Error! Bookmark not defined.

LAB 1: Introduction to VHDL and Altera Quartus with the design of Full Adder and reproduce the output of FPGA board

Objectives

Part 1

- *Comprehend* the concept of hardware descriptive languages in general and VHDL in particular.
- *Illustrate* software tool Quartus with the design of full adder
- *Produce* the result using hardware FPGA by loading the bit file.

Part 2

This part is used to implement couple of basic logic functions in VHDL as well as half adder and full adder design. These designs will be simulated and synthesize on Quartus.

Part 1 -Familiarize yourself with VHDL and Quartus:

Introduction to Quartus:

Altera Quartus II is design software produced by [Altera](#). Quartus II enables analysis and synthesis of [HDL](#) designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Quartus includes an implementation of [VHDL](#) and [Verilog](#) for hardware description, visual editing of logic circuits, and vector waveform simulation.

Introduction to VHDL

VHDL is a hardware description language. It described the behavior of an electronic circuit or system, from which the physical circuit or system can then be attained (implemented). VHDL stands for VHSIC Hardware Description Language. VHSIC is itself an abbreviation for Very High-Speed Integrated Circuits, an initiative funded by US DOD in the 1980s that led to the creation of VHDL. VHDL is intended for circuit synthesis as well as circuit simulation. However, though VHDL is fully simulated, not all constructs are synthesizable.

A fundamental motivation to use VHDL (or its competitor, Verilog) is that VHDL is a standard, technology/vendor independent language, and is therefore portable and reusable. The two main immediate applications of VHDL are in the field of Programmable Logic Devices (including CPLDs and FPGAs) and ASICs. A final note regarding VHDL is that contrary to regular computer programs which are sequential, its statements are inherently concurrent (parallel). In VHDL, only statements placed inside a *Process*, *Function* or *Procedure* are executed sequentially.

As mentioned above, one of the major utilities of VHDL is that it allows the synthesis of a circuit or system in a programmable device (PLD or FPGA) or in an ASIC. The steps followed during such a project are summarized in figure 1. 1

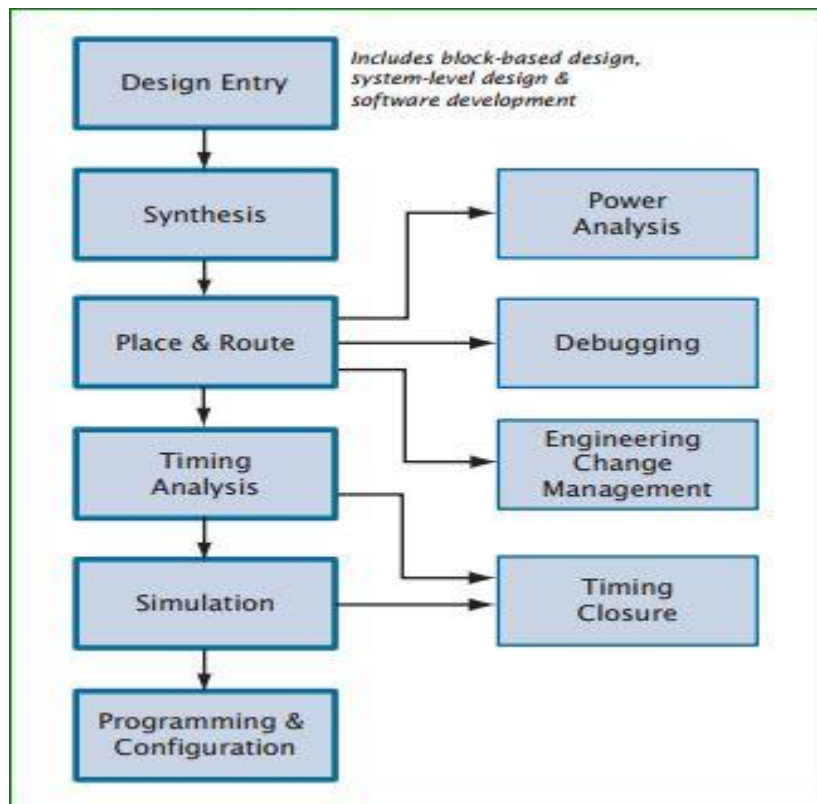


Figure 1.1 Quartus II Design Flow

Fundamental VHDL Units

Library declaration: Contains a list of all libraries to be used in the design. For example: ieee, std, work etc. Their declarations are as follows

```
Library ieee;
```

```
USE ieee.std_logic_1164.all;
```

```
USE ieee.std_logic_arith.all;
```

Entity:

VHDL files are basically divided into two parts, the entity and the architecture. The entity is basically where the circuits (in & out) ports are defined. There is a multitude of I/O ports available, but this lab will only deal with the two most usual ones, the INput and OUTput ports. (Other types of ports are for example the INOUT and BUFFER ports.) The entity of the circuit in figure 1.2 should look something like below. Please notice that comments in the code are made with a double-dash (--).

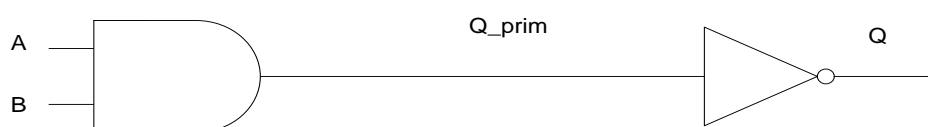


Figure 1.2 NAND gate using AND & NOT gate

```

ENTITY nandgate IS
PORT(
A: IN BIT;
B: IN BIT;
Q: OUT BIT -- Note! No ';' here!
);
END nandgate;

```

Now that we have defined the I/O interface to the rest of the world for the NAND gate, we should move on to the architecture of the circuit.

Architecture:

The entity told us nothing about how the circuit was implemented, this is taken care of by the architecture part of the VHDL code. The architecture of the NAND gate matching the entity above could then be written as something like this...

```

ARCHITECTURE dataflow OF nandgate IS
SIGNAL Q_prim: BIT;
BEGIN
Q_prim <= A AND B; -- The AND-operation.
Q <= NOT Q_prim; -- The inverter.
END dataflow;

```

Behavior modeling, Processes

We can write a behavior architecture body of an entity which describes the function in an abstract way. Such an architecture body includes only process statements.

Processes are used:

- For describing component behavior when they cannot be simply modeled as delay elements.
- To model systems at high levels of abstraction.

Process contains:

Conventional programming language constructs. A process is a sequentially executed block of code, which contains.

- Evaluating expressions.
- Conditional execution.
- Repeated execution.
- Subprogram calls.
- Variable assignments, e.g., $x := y$, which, unlike signal assignment, take effect immediately.
- if-then-else and loop statements to control flow, and
- Signal assignments to external signals.

Notes:

1. Signal assignment statements specify the new value and the time at which the signal is to acquire this value. The textual order of the concurrent signal assignment statements (CSAs) do NOT effect the results.
2. Processes contain sensitivity lists in which signals are listed, which determine when the process executes.
3. In reality, CSAs are also processes without the process, begin and end keywords.

Structure modeling

Structural model: A description of a system in terms of the interconnection of its components, rather than a description of what each component does. A structural model does NOT describe how output events are computed in response to input events.

A VHDL structural description must possess:

- The ability to define the list of components.
- The definition of a set of signals to be used to interconnect them.
- The ability to uniquely label (distinguish between) multiple copies of the same component.

Part 2 –Design and Implementation of Four-bit adder:

In this lab you will design, test, and simulate a basic logic circuit using the Quartus II development software. The circuit you will create is a four-bit adder using both behavioral and structural VHDL coding. Below is a schematic diagram of the complete circuit.

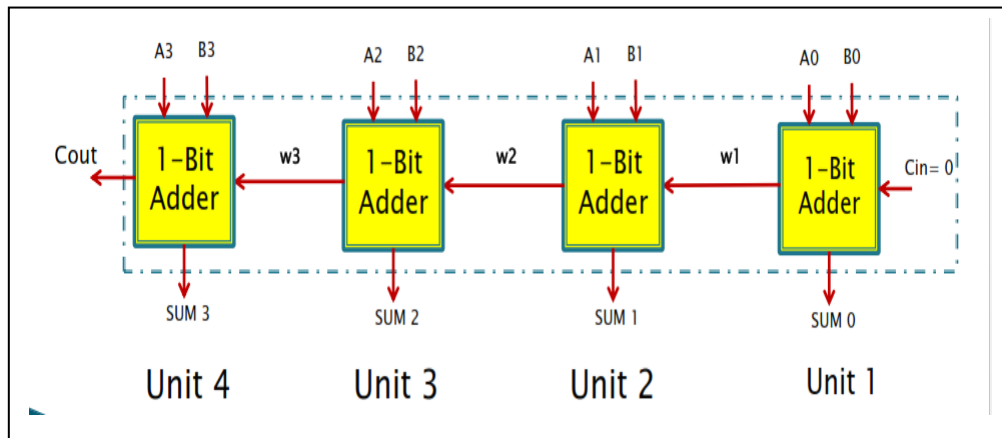


Figure 1.3 4- bit adder

Below is a detailed diagram of the full adder circuit used. It consists of three two-input AND gates, two XOR gates, and one three-input OR gate. The individual gates will be coded separately using behavioral style VHDL, and then stitched together using structural style VHDL to create the full adder.

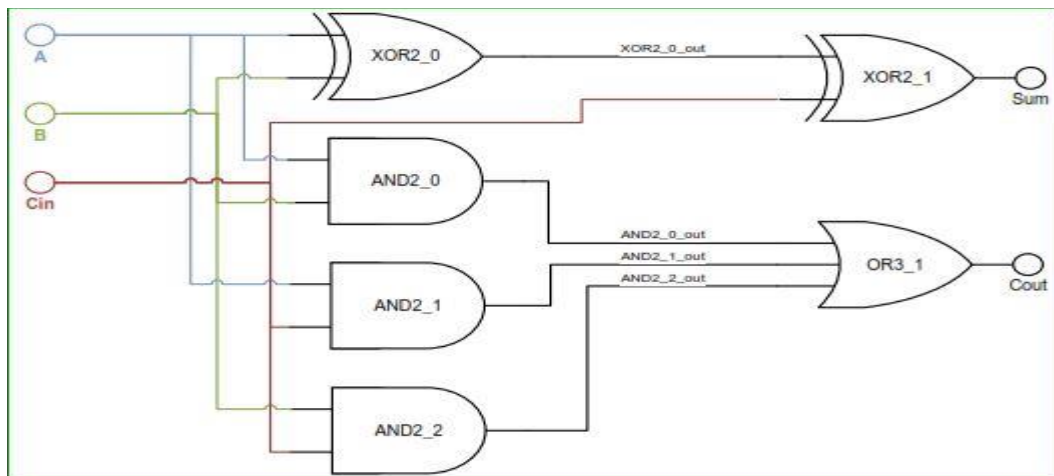


Figure 1.4 4- bit adder gate-level

Task 1: Create a New Project

1. Start the Quartus II software.

From the Windows Start Menu, select:

All Programs → Other Apps → Altera → Quartus II 13.1 → Quartus II 13.1 (32-Bit)

2. Start the New Project Wizard.

If the opening splash screen is displayed, select: **Create a New Project (New Project Wizard)**, otherwise from the Quartus II Menu Bar select: **File → New Project Wizard**.

3. Select the Working Directory and Project Name.

Working Directory	H:\Altera_Training\Lab1
Project Name	Lab1
Top-Level Design Entity	Lab1

Click **Next** to advance to page 2 of the New Project Wizard.

*Note: A window may pop up stating that the chosen working directory does not exist. Click **Yes** to create it.*

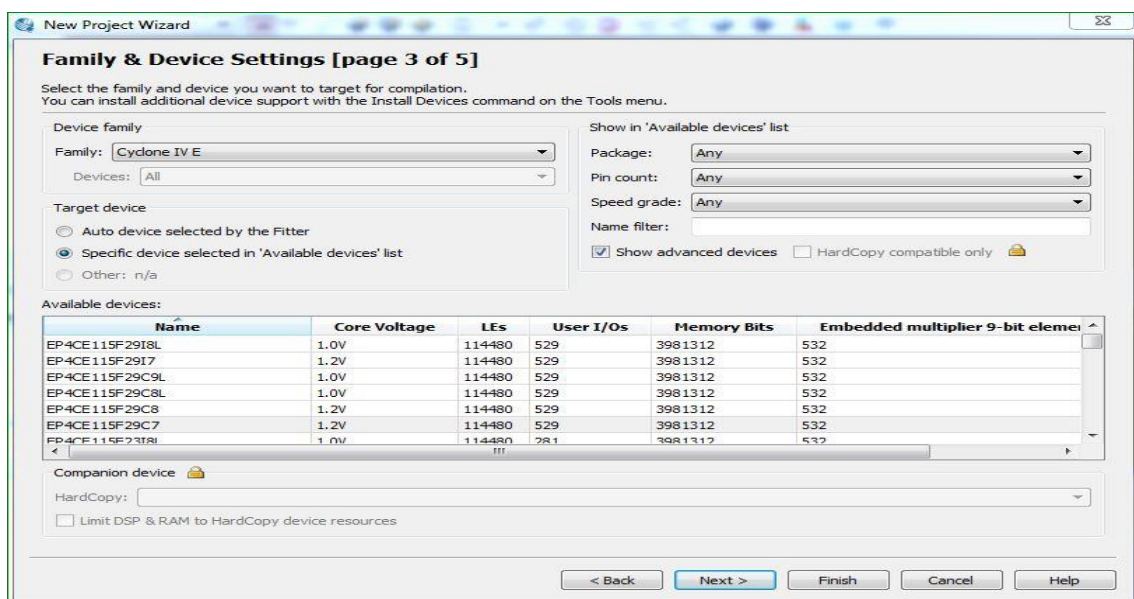
4. Click **Next** again as we will not be adding any preexisting design files at this time.

5. Select the family and Device Settings.

From the pull-down menu labeled **Family**, select **Cyclone IV E**.

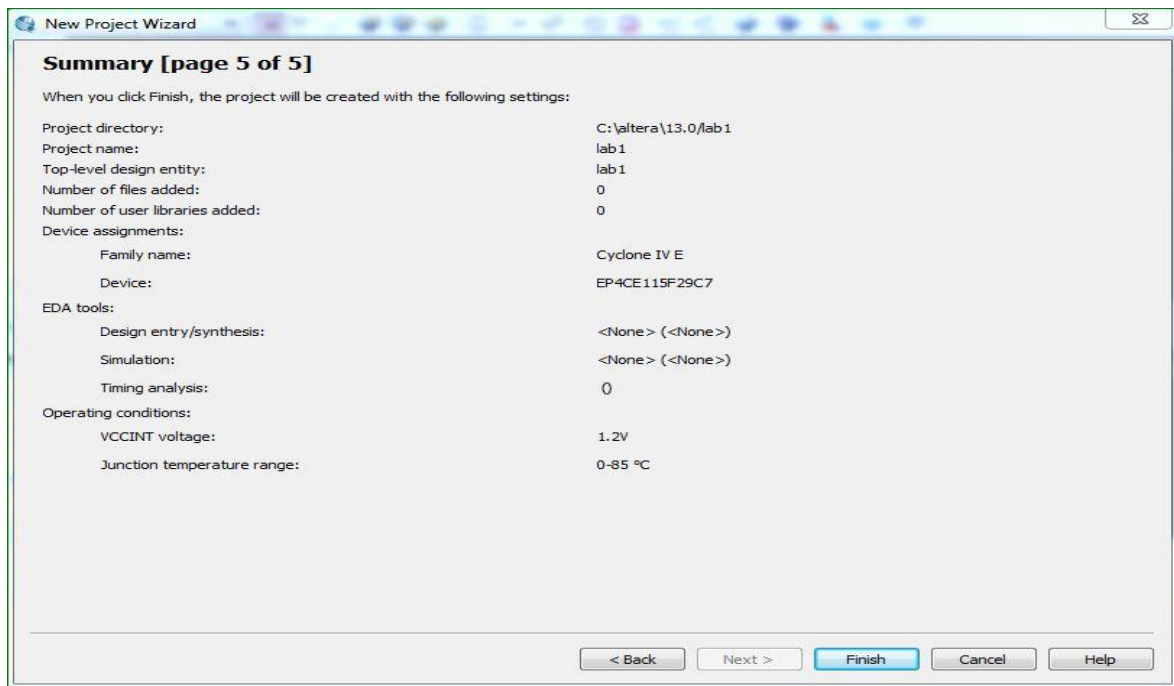
In the list of available devices, select **EP4CE115F29C7N**.

Click **Next**.



6. Click **Next** again as we will not be using any third-party EDA tools

7. Click **Finish** to complete the New Project Wizard

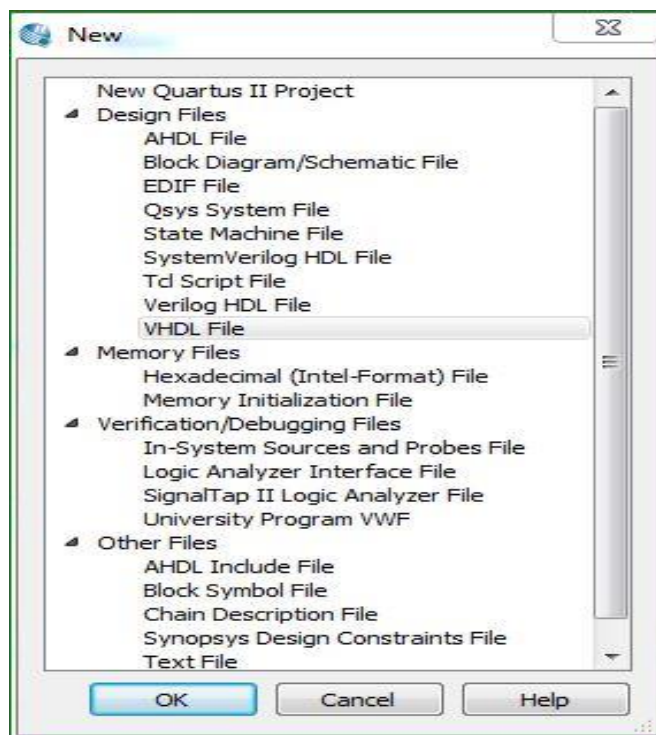


Task 2: Create, Add, and Compile Design Files

1. Create a new Design File.

Select: **File** → **New** from the Menu Bar.

Select: **VHDL File** from the **Design Files** list and click **OK**.



- Copy and paste the following code into your new VHDL file, then save it by selecting **File** → **Save**. Name the file **bitadder** and click **Save** in the **Save As** dialog box.

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
entity bitadder is
port (A: in std_logic;
      B: in std_logic;
      C: in std_logic;
      sum: out std_logic;
      cout : out std_logic);
end bitadder;
architecture behavioral of bitadder is
begin
  sum<= A xor B xor C;
  cout<= (A and B) or (A and C) or (B and c);
end behavioral;
```

- Create another new VHDL file following the directions in step 1 of task 2.
- Copy and paste the following code into your new VHDL file, then save it by selecting **File** → **Save**. Name the file **adder4** and click **Save** in the **Save As** dialog box.

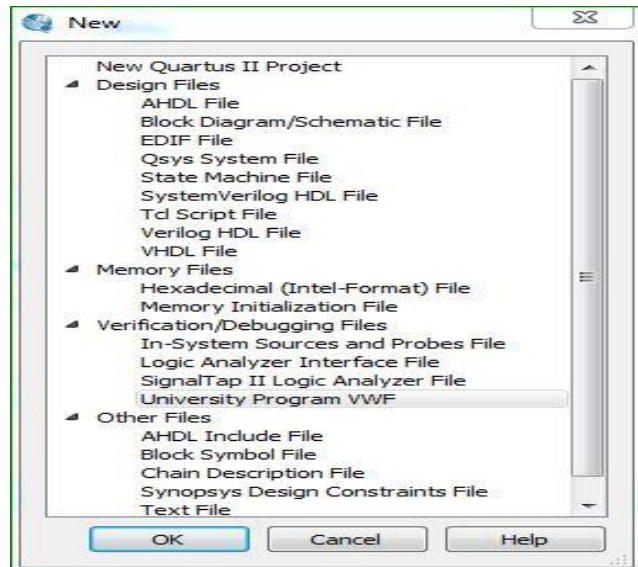
```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
entity adder4a is
port (A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      C: in std_logic;
      sum: out std_logic_vector(3 downto 0);
      cout : out std_logic);
end adder4a;
architecture behavioral of adder4a is
  component bitadder is
  port (A: in std_logic;
        B: in std_logic;
        C: in std_logic;
        sum: out std_logic;
        cout : out std_logic);
  end component;
  signal w1, w2, w3: std_logic;
begin
  unit1: bitadder port map (A(0), B(0), '0', sum(0), w1);
  unit2: bitadder port map (A(1), B(1), w1, sum(1), w2);
  unit3: bitadder port map (A(2), B(2), w2, sum(2), w3);
  unit4: bitadder port map (A(3), B(3), w3, sum(3), cout);
end behavioral;
```

Task 3: Simulate Design using Quartus II

1. Create a Vector Waveform File (vwf)

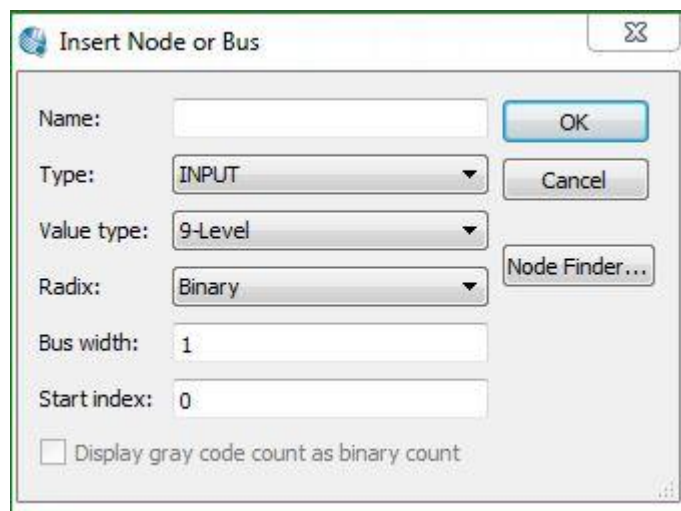
Click the **New File** icon on the Menu Bar 

Select **Vector Waveform File** under the **Verification/Debugging Files** heading.



2. Add Signals to the Waveform

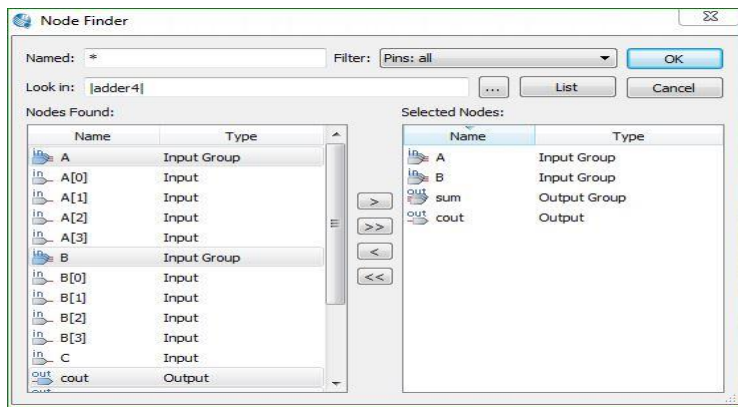
Select **Edit** → **Insert** → **Insert Node or Bus...** to open the **Insert Node or Bus** window.



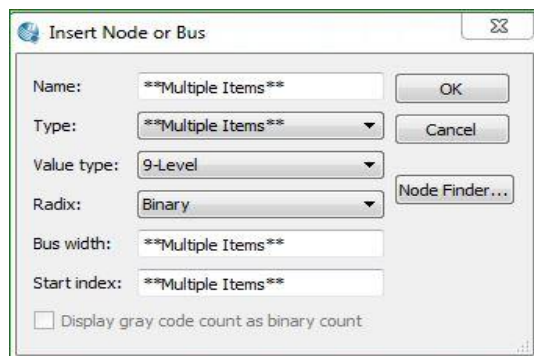
3. Click the **Node Finder...** button in the **Insert Node or Bus** window to open the **Node Finder**

window. From the **Filter** drop-down menu, select **Pins: all** and click the **List** button. This will display all the inputs and outputs to the circuit.

Highlight all the **Input Groups A, B**, and **Output Group Sum** in the **Node Finder** window and click the right arrow  to select them. Then click **OK** to close the **Node Finder** window.

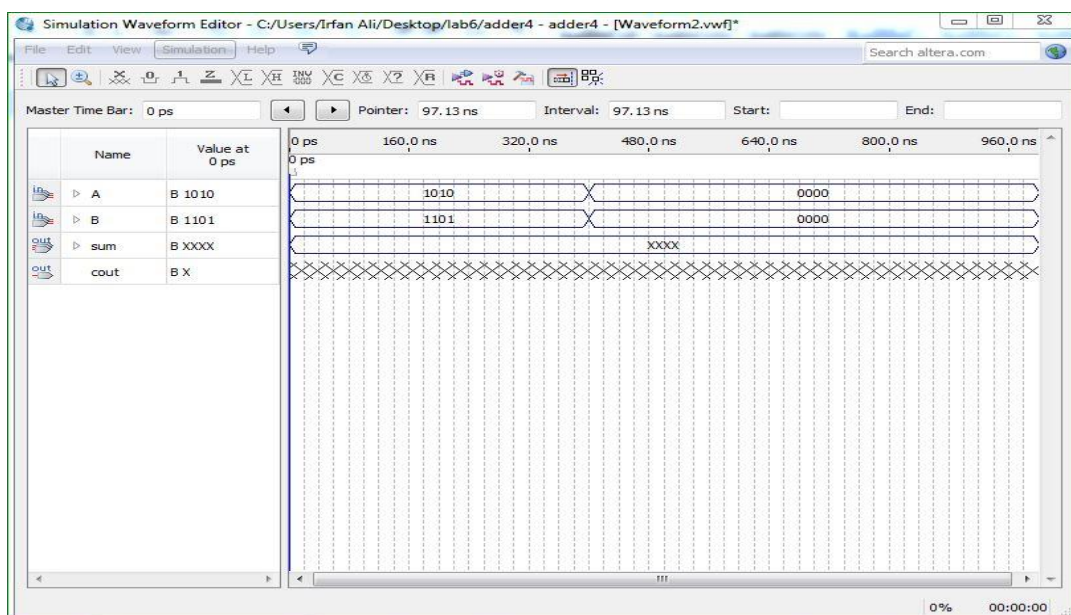



4. In the **Insert Node or Bus** window, change **Radix** to **Binary**. This will make it easier for us to enter values and interpret the results.



5. Input waveforms can be drawn in different ways. The most straightforward way is to indicate a specific time range and specify the value of a signal. To illustrate this approach, click the mouse on the A waveform near the 0-ns point and then drag the mouse to the 400-ns point. The selected time interval will be highlighted in blue. Press **Ctrl + Alt + B** to open the **Arbitrary Value** window to enter the values manually.

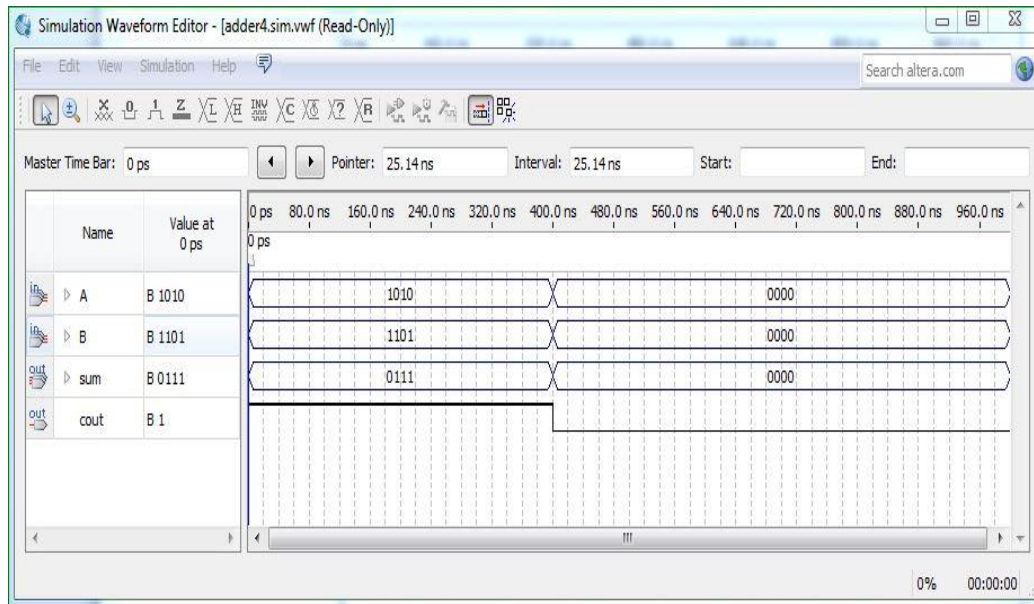
We will use this approach to create the waveform for B, which should change from 0 to 400 ns.



Leave **Sum** with a value of undefined (X), as values for this will be generated when the simulation runs. Click the **Save** icon  on the Menu bar and save the vwf file with the filename **adder4.vwf**.

6. In the Main window, select *Simulation / Options* and then select *Quartus II Simulator*. Select *OK*.
7. In the Main window, select *Simulation* and then select *Run Functional Simulation*.
8. Now you should see your simulation output with the outputs defined.

Note: The file will indicate "read-only" meaning you can't edit it.



Task 4: Implementing the Design on the DE2 Board

1. From the Menu Bar select: **Assignments** → **Assignment Editor**
2. Double-click <<new>> in the **To** column and select **Node Finder** button, List down all the input and output pins and click OK.




Assignment Editor									
<<new>>		Filter on node names: *						Category: All	
tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag	
1	<<new>>	<<new>>	<<new>>						

3. Double-click the adjacent cell **Assignment Name** and select **Location (Accepts wilds cards/groups)**.

Double-click the adjacent cell **Value** and assign the pin number.

Continue to assign the pins as seen in the table below.

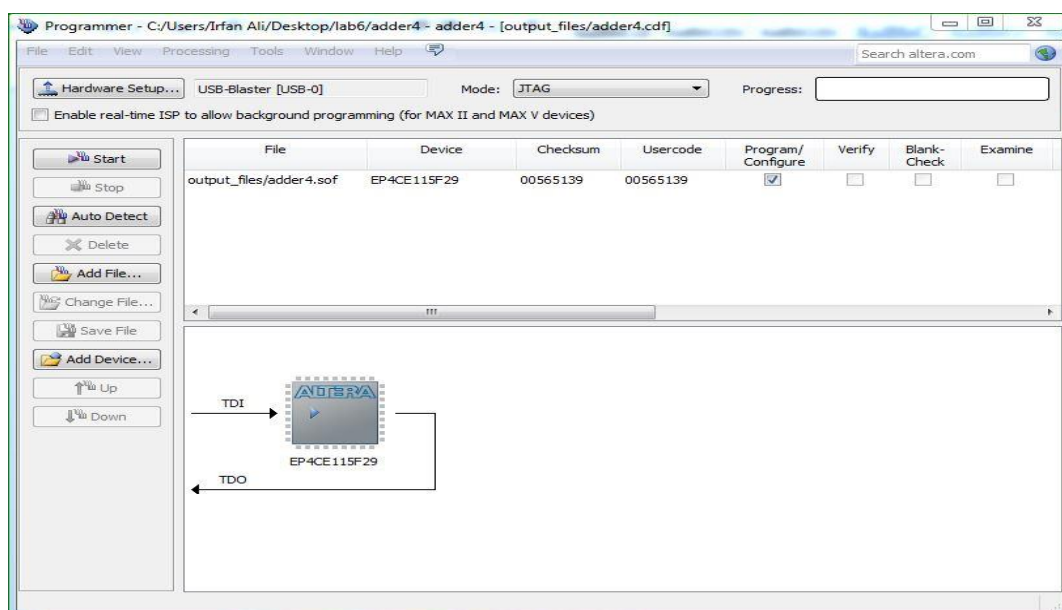
	To	Location	DE2 Board Description
1	A[0]	PIN_AB28	SW0
2	A[1]	PIN_AC28	SW1
3	A[2]	PIN_AC27	SW2
4	A[3]	PIN_AD27	SW3
5	B[0]	PIN_AB27	SW4
6	B[1]	PIN_AC26	SW5
7	B[2]	PIN_AD26	SW6
8	B[3]	PIN_AB26	SW7
10	Sum[0]	PIN_G19	LED0
11	Sum[1]	PIN_F19	LED1
12	Sum[2]	PIN_E19	LED2
13	Sum[3]	PIN_F21	LED3
14	Cout	PIN_F18	LED4

- Save the pin assignments by selecting **File** → **Save** from the Menu Bar, or by clicking the Save button  on the toolbar.
- Compile the design again by clicking the Start Compilation button on the toolbar .
- Plug in and power on the DE2 board. Make sure that the **RUN/PROG Switch for JTAG/AS Modes** is in the **RUN** position.
- In the Quartus II window click the **Programmer** button on the Toolbar to open the Programmer window .

The **Hardware Setup...** must be **USB-Blaster [USB-0]**. If not, click the **Hardware Setup...** button and select **USB-Blaster [USB-0]** from the drop-down menu for **currently selected hardware**.

Mode should be set to **JTAG**.

Make sure that the **File** is **adder4.sof**, **Device** is **EP4CE115F29C7N** and the **Program/Configure** box is checked.



Then click the **Start** button to program the DE2 board. When the progress bar reaches 100%, programming is complete.

8. You can now test the program on the DE2 board by using the toggle switches located along the bottom of the board.

SW0 through SW3 are the four bits of data for A.

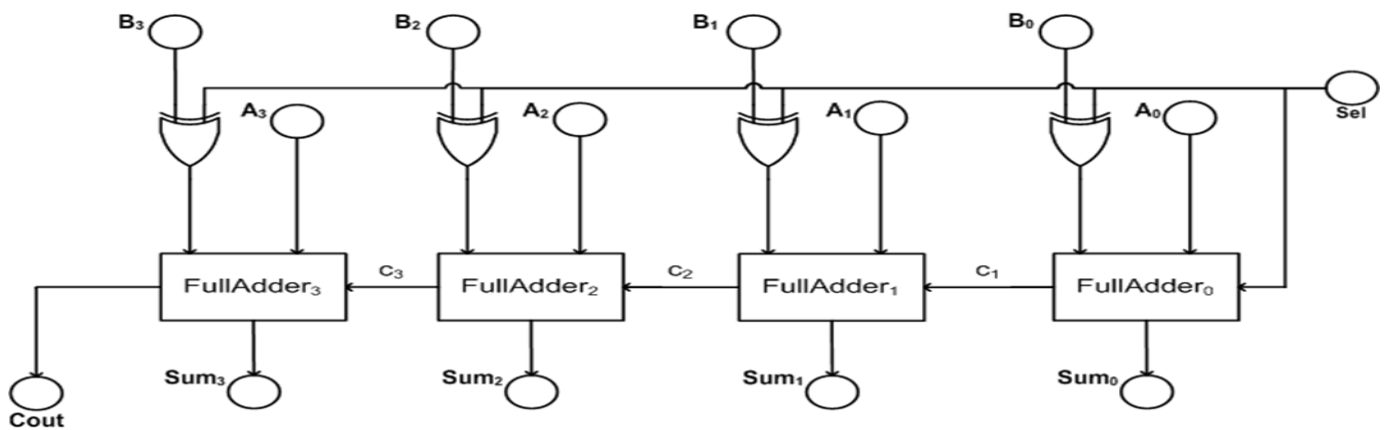
SW4 through SW7 are the four bits of data for B.

The output of the operation is displayed on the first four LEDs (LED0-LED3) located above the blue push buttons on the DE2 board.

LED4 is the indicator for Cout.

Practice Tasks: Transfer VHDL code into the FPGA board

Use below diagram to write a VHDL code for a 4-bit Subtractor/Adder. Show the waveform.



Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 2: To design the combinational and sequential circuit components using VHDL and *reproduce* the output on the FPGA board

Objectives

Part 1

- This lab session *demonstrates* the concept of combinational logic problems using VHDL. -
- Quartus shall be *used* for the compilation and functional verification of VHDL *design* files.

Part 2

- This lab session *demonstrates* the concept of sequential logic problems using VHDL.
- Quartus shall be *used* for the compilation and functional verification of VHDL *design* files.

Part 1 –Design and Implementation of Combinational Logic

Task 1: Design and simulation of combinational warning system of an automobile

Warning output is 1 when ignition is on and either door is open, or seat belt is not fastened or both latter two conditions are true.

architecture structural of Alarm is

-- Declarations

component AND2a

port (in1a, in2b: **in** std_logic;

out1a: **out** std_logic);

end component;

component OR2a

port (ina1, in2b: **in** std_logic;

out1a: **out** std_logic);

end component;

component NOT1a

port (in1a: **in** std_logic;

out1a: **out** std_logic);

end component;

-- declaration of signals used to interconnect gates

signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;

begin

-- Component instantiations statements

U0: NOT1a **port map** (DOOR, DOOR_NOT);

U1: NOT1a **port map** (SBELT, SBELT_NOT);

U2: AND2a **port map** (IGNITION, DOOR_NOT, B1);

U3: AND2a **port map** (IGNITION, SBELT_NOT, B2);

U4: OR2a **port map** (B1, B2, WARNING);

end structural;

Process Syntax in VHDL

- Syntax

<label> : process(sensitivity list)

begin

<statements>

end process;

- Sensitivity List
 1. Defines what a process is dependent on
 2. List of signals in the system
 3. Process will execute when any of these signals change
 4. Can use constructs such as if statements in a process

Case statements

The case statement executes one of several sequences of statements, based on the value of a single expression. The syntax is as follows,

```
case expression is
when choices =>
  sequential statements
when choices =>
  sequential statements
-- branches are allowed
[ when others => sequential statements ]
end case;
```

If Statements

The if statement executes a sequence of statements whose sequence depends on one or more conditions.

The syntax is as follows:

```
if condition then
  sequential statements
[elsif condition then
  sequential statements ]
[else
  sequential statements ]
end if;
```

Task2: Design and test a 4-1 MUX using ‘CASE’ statement

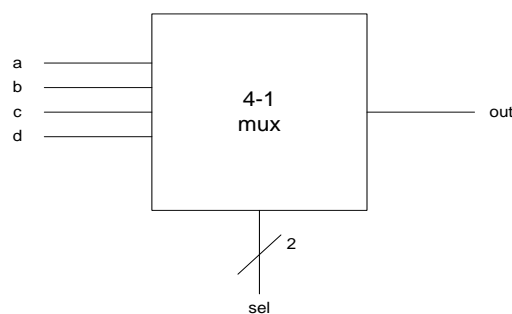


Figure 2.1 Block Diagram of 4-1 multiplexer

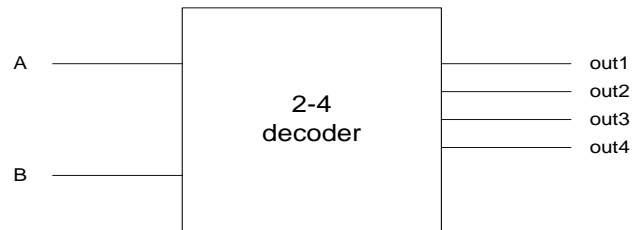
Task 3: Design and test a 2-4 decoder

Figure 2.2 Block Diagram of 2-4 decode

Table 2.1 Operators in VHDL

Arithmetic Operators Symbol Function + Addition - Subtraction & Concatenation NOTE + and - are also signs example: signal A: bit_vector(5 downto 0); signal B,C: bit_vector(2 downto 0); B <= '0' & '1' & '0'; C <= '1' & '1' & '0'; A <= B & C; -- A now has "010110"	Assignment Operators Symbol Function <= Assignment of signals := Assignment of variables, also assignment of signals at declaration NOTE <= is also a relational operator
Logic Operators Symbol Function and nand or nor xor xnor	Multiplying operators Symbol Function * Multiplication / Division mod Modulus rem Reminder
Sign Operators Symbol Function	Symbol Function ** Exponentiation

+ Plus - Minus NOTE + and - are also adding operators <p style="text-align: center;">Relational operators</p> Symbol Function = Equal /= Not equal < Less than <= Less than or equal > Greater than >= Greater than or equal NOTE <= is also an assignment operator	abs Absolute value not takes complement of value <p style="text-align: center;">Shift operators</p> Symbol Function rol Rotate left logical ror Rotate right logical sla Shift left arithmetic sra Shift right arithmetic sll Shift left logical srl Shift right logical
---	--

Part 2 –Design and Implementation of Sequential Logic

In the previous part, problems related to multiplexer, decoder, comparator and implementation of small components were discussed in detail. Also new statements related to behavioral modeling like case and if-then-else were discussed too. Now in this lab new statements like “with-select”, “loops” and “generate” will be discussed and you have to write the VHDL code and also the verification (test bench code) for each task using few of these statements.

Sequential Statements

There are six variants of the sequential statement, namely

- PROCESS Statement
- IF-THEN-ELSE Statement
- CASE Statement
- LOOP Statement
- WAIT Statement
- ASSERT Statement

Concurrent Statements

- Signal Assignment Statement
 - Simple Assignment Statement
 - Selected Assignment Statement
 - Conditional Assignment Statement
- Component Instantiation Statement
- Generate Statement
- Assert Statement

Select Statements

Selected signal assignment statement is used to set the values of a signal to one of several alternatives based on a selection criterion.

Syntax:

```
WITH discriminant SELECT
target_signal <= value1 WHEN choice1,
value2 WHEN choice2,
valueN WHEN choiceN [or OTHERS];
```

Conditional Assignment Statement

The format is

```
target_signal <= value1 WHEN condition1 ELSE
value2 WHEN condition2 ELSE
```

...

```
valueN-1 WHEN conditionN-1 ELSE
valueN;
```

Example:

```
ARCHITECTURE data_flow OF two_1_mux is
BEGIN
output <= A WHEN Control = '0' ELSE
'B';
END data_flow;
```

Loop Statement

A loop statement consists of a loop header before the key word LOOP and a loop body which appears between LOOP and END LOOP. The loop header determines whether it concerns the indefinitely loop, conditioned loop or counted loop. The loop body contains a list of statements to be executed in each iteration.

```
[ loop_label :] loop
sequential statements
[next [label] [when condition];
[exit [label] [when condition];
end loop [ loop_label];
```

for-Loop statement

The for-loop uses an integer iteration scheme that determines the number of iterations. The syntax is as follows,

```
[ loop_label :] for identifier in range loop
sequential statements
[next [label] [when condition];
[exit [label] [when condition];
end loop[ loop_label ];
```

Examples

```
Loop_1: for count_value in 1 to 10 loop
exit Loop_1 when reset = '1';
A_1: A(count_value) := '0';
end loop Loop_1;
A_2: B <= A after 10 ns;
```

If the condition_1 in the iteration count_value is TRUE, then the next statement will be executed. The next statement to be executed will be Assign_1 in the iteration count_value+1. Otherwise, the sequence of operations is as specified, i.e. Assign_2 is executed.

Next and Exit Statement

The next statement skips execution to the next iteration of a loop statement and proceeds with the next iteration. The syntax is

```
next [label] [when condition];
```

The **when** keyword is optional and will execute the next statement when its condition evaluates to the Boolean value TRUE.

The exit statement skips the rest of the statements, terminating the loop entirely, and continues with the next statement after the exited loop. The syntax is as follows:

exit [*label*] [**when** *condition*];

Wait statement

The wait statement will halt a process until an event occurs. There are several forms of the wait statement,

wait until *condition*;

wait for *time expression*;

wait on *signal*;

wait;

Generate Statement

Generate Statements: describe regular and/or slightly irregular structure by automatically generating component instantiations instead of manually writing each instantiation.

There are two variants of the generate statement:

- **FOR GENERATE statement**

Provides a convenient way of repeating either a logic equation or a component instantiation.

- **IF GENERATE statement**

Example

```
ARCHITECTURE bus16_wide OF reg16 IS
```

```
COMPONENT dff
```

```
PORT ( d, clk : IN STD_LOGIC,
```

```
q : OUT STD_LOGIC);
```

```
END COMPONENT;
```

```
BEGIN
```

```
G1 : FOR i IN 0 to 15 GENERATE
```

```
dff1: dff PORT MAP (input (i), clock, output(i));
```

```
END GENERATE G1;
```

```
END bus16_wide;
```

Task 1: Design of a Latch

A latch is transparent when the control signal is high. The latch is simply controlled by enable bit but has nothing to do with clock signal. However, control signal can also be clock.

Code:

```
-----
-- Simple D Latch
```

```
-- notice this difference from flip-flops
```

```
-----
library ieee ;
```

```
use ieee.std_logic_1164.all;
```

```
-----
entity D_latch is
```

```
port(  data_in:in std_logic;
```

```
      enable:      in std_logic;
```

```

        data_out:      out std_logic
    );
end D_latch;
-----

```

```

architecture behv of D_latch is
begin
    -- compare this to D flipflop
    process(data_in, enable)
    begin
        if (enable='1') then
            -- no clock signal here
            data_out <= data_in;
        end if;
    end process;
end behv;

```

Task 2: Design of D Flip-Flop

Flip-flop is the basic component in sequential logic design. The flip-flop is unique in that it stores the input on the **clock event**. The logic symbol for this flip-flop is given in fig 2.3

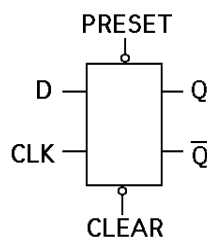


Figure 2.3 D-Flip Flop

```

-----
-- we assign input signal to the output
-- at the clock rising edge
-----

```

```

library ieee ;
use ieee.std_logic_1164.all;
use work.all;
-----

```

```

entity dff is
port(   data_in:in std_logic;
        clock:  in std_logic;
        data_out:out std_logic
    );
end dff;
-----

```

```

architecture behv of dff is
begin
    process(data_in, clock)
    begin
        -- clock rising edge
        if (clock='1' and clock'event) then
            data_out <= data_in;
        end if;
    end process;
end behv;

```

```

    end if;
  end process;
end behv;

```

Task 3: Design of Shift Register

The main usage for a shift register is for converting from a serial data input stream to a parallel data output or vice versa. For a serial-to-parallel data conversion, the bits are shifted into the register at each clock cycle, and when all of the bits (usually eight bits) are shifted in, the 8-bit register can be read to produce the eight bit parallel output.

Shift registers are a type of sequential logic circuit, mainly for storage of digital data.

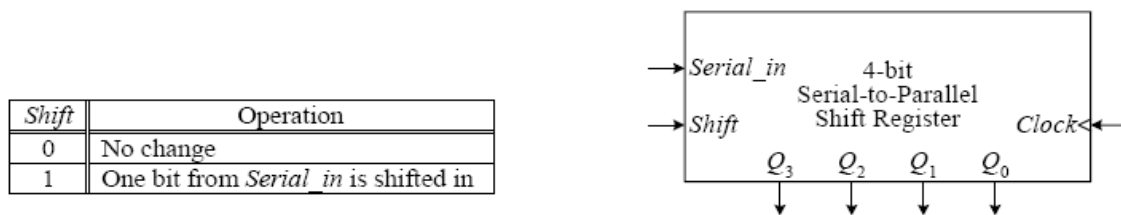


Figure 2.4 Serial to Parallel Shift register

- ✓ *The code given below is for simple shift register with single-bit output. Convert this code for Serial-to-Parallel Shift Register*

Code:

```

-----
-- 4-bit Shift-Register/Shifter
-----
library ieee ;
use ieee.std_logic_1164.all;
-----
entity shift_reg is
port(   Serial_In:      in std_logic;
        clock:          in std_logic;
        shift:          in std_logic;
        Q:              out std_logic);
end shift_reg;
-----
architecture behv of shift_reg is
  -- initialize the declared signal
  signal S: std_logic_vector(3 downto 0):="1101";
begin
  process(Serial_In, clock, shift, S)
  begin
    -- everything happens upon the clock changing
    if clock'event and clock='1' then
      if shift = '1' then
        S <= Serial_In & S(3 downto 1) -- is it shift left or right. Also write the converse.
      end if;
    end if;
  end process;
  -- concurrent assignment
  Q <= S(0);
end behv;

```

Practice Tasks: Transfer VHDL code into the FPGA board:

1. **(3 bit counter)** Implement a counter using clock counts from 0 to 8. If the reset=1 the count becomes zero else if clock is rising and enable is pressed (enable=1) it counts from zero to 8. On reaching 8 it again counts from zero.
2. **(integer counter)** Make some changes in the above code to implement integer counter that counts from 0 to 512 and use a built in function to convert integer value into std_logic. Make count equal to zero when count reaches 512.
Hint: variable m: integer range 0 to 511 := 0; and before ending process use function to convert integer to std_logic type.

The conv_std_logic_vector function

signal i1, i2 : integer;

v3 <= conv_std_logic_vector(i1, 4); -- = "1101"

v4 <= conv_std_logic_vector(i2, 4); -- = "1101"

3. Implement a 4-bit shift register using loop statement.
4. Design and test a circuit that divides a clock signal by 6. Simulation result should match the following result.

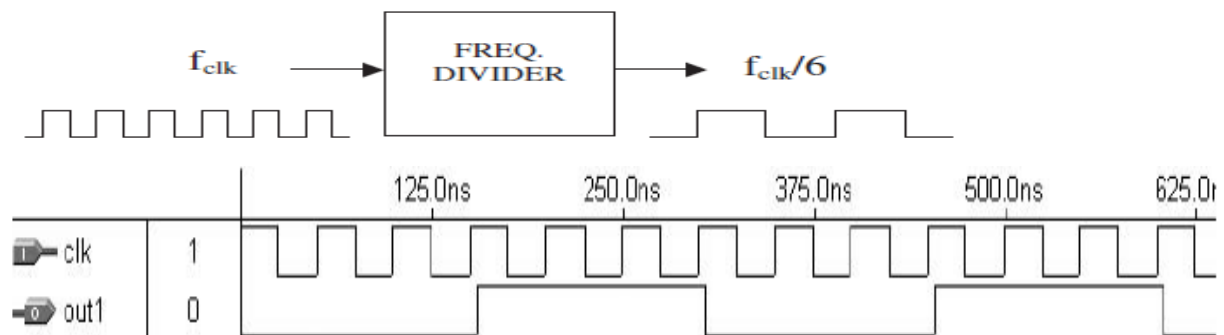


Figure 2.5 clock divider desired simulation result

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 3: To design the finite state machines using VHDL and reproduce the results on FPGA board

Objectives

The objective of this lab is to *design* the finite state machines using VHDL

Part 1: *Design* and testing of Mealy FSM.

Part 2: *Design* and testing of Moore FSM.

Part 1 – Introduction, specification, design and testing of Mealy FSM

There are two basic types of sequential circuits: Mealy and Moore. Because these circuits transit among a finite number of internal states, they are referred to as finite state machines (FSMs). In a Mealy circuit, the outputs depend on both the present inputs and state. In a Moore circuit, the outputs depend only on the present state. The most common way of schematically representing a Mealy sequential circuit is shown in Fig.3.1

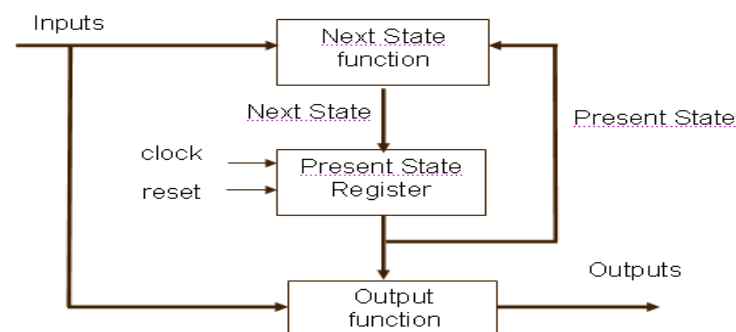


Figure 3.1 Generalized Mealy sequential circuit

The state register normally consists of D flip-flops (DFFs). However, other types of flip-flops can be utilized, such as JKFFs. The normal sequence of events is: (1) inputs X change to a new value, (2) after a clock period delay, outputs Z and next state NS become stable at the output of the combinational circuit, the next state signals NS are stored in the state register; that is, next state NS replace present state PS at the output of the state register, which feeds back into the combinational circuit. At this time, a new cycle is ready to start. These operational cycles are synchronized with the clock signal CLK .

It is worth mentioning that some authors further classify sequential circuits into two categories. The first category, referred to as “regular sequential circuits”, includes circuits like (shift) registers, FIFOs, and binary counters and variants. The second category, referred to as “finite state machines” (FSMs), include circuits that typically do not exhibit a simple, repetitive pattern.

Enumerated Types in VHDL

An enumerated type consists of lists of character literals or identifiers. The enumerated type can be very handy when writing models at an abstract level. The syntax for an enumerated type is,

type type_name is (identifier list or character literal);

Here are some examples,

type my_3values is ('0', '1', 'Z');

type PC_OPER is (load, store, add, sub, div, mult, shiftr, shiftr);

type hex_digit is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F');

type state_type is (S0, S1, S2, S3);

Examples of objects that use the above types:

signal SIG1: my_3values;

variable ALU_OP: pc_oper;

variable first_digit: hex_digit := '0';

signal STATE: state_type := S2;

Example 1: MEALY machine design – BCD to Excess-3 code converter

In this example, we'll design a serial converter that converts a binary coded decimal (BCD) digit to an excess-3-coded decimal digit. Excess-3 binary-coded decimal (XS-3) code, also called biased representation or Excess-N, is a complementary BCD code and numeral system. It was used on some older computers with a pre-specified number N as a biasing value. It is a way to represent values with a balanced number of positive and negative numbers. In our example, the XS-3 code is formed by adding 0011 to the BCD digit. The table and state graph in Fig.3.2 describe the functionality of our design.

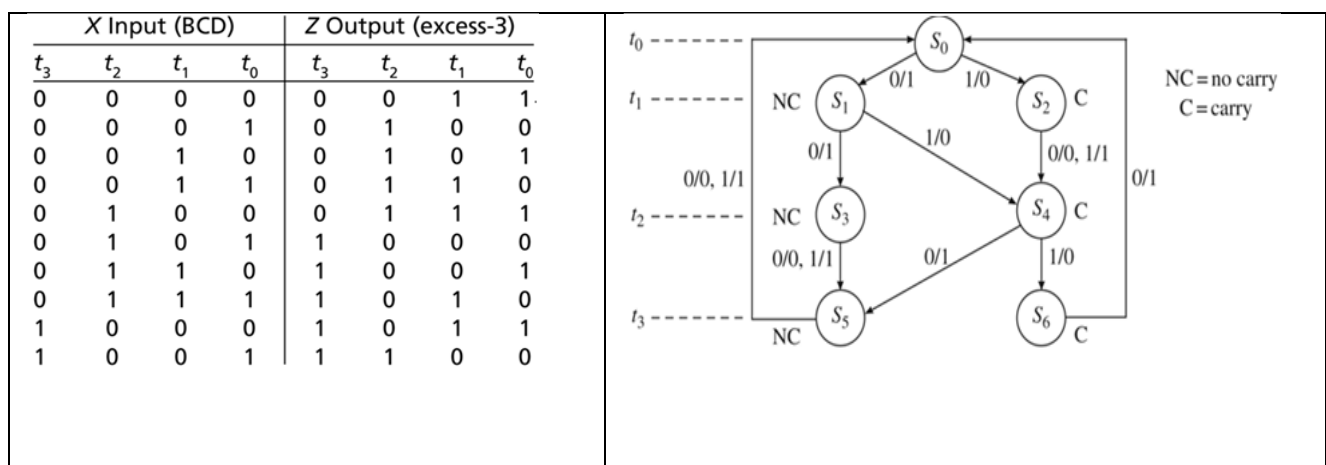


Figure 3.2 Truth Table and Finite State machine of BCD to Excess-3 code converter

There are several ways to model this sequential machine. One popular/common approach is to use **two processes** to represent the two parts of the circuit: the combinational part and the state register. For clarity and flexibility, we use VHDL's *enumerated data type* to represent the FSM's states. The following VHDL code describes the converter.

```
-- Behavioral model of a Mealy state machine: code converter w/ 2 processes
-- It is based on its state table. The output (Z) and next state are
-- computed before the active edge of the clock. The state change occurs on the rising edge of the clock.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Code_Converter is
port(
enable: in std_logic;
X, CLK: in std_logic;
Z: out std_logic);
end Code_Converter;
architecture Behavioral of Code_Converter is
type state_type is (S0, S1, S2, S3, S4, S5, S6);
signal State, Nextstate: state_type;
-- a different way: represent states as integer signals:
-- signal State, Nextstate: integer range 0 to 6;
begin
-- Combinational Circuit
process(State, X)
```

```

begin
case State is
when S0 =>
if X = '0' then Z <= '1'; Nextstate <= S1;
else Z <= '0'; Nextstate <= S2; end if;
when S1 =>
if X = '0' then Z <= '1'; Nextstate <= S3;
else Z <= '0'; Nextstate <= S4; end if;
when S2 =>
if X = '0' then Z <= '0'; Nextstate <= S4;
else Z <= '1'; Nextstate <= S4; end if;
when S3 =>
if X = '0' then Z <= '0'; Nextstate <= S5;
else Z <= '1'; Nextstate <= S5; end if;
when S4 =>
if X = '0' then Z <= '1'; Nextstate <= S5;
else Z <= '0'; Nextstate <= S6; end if;
when S5 =>
if X = '0' then Z <= '0'; Nextstate <= S0;
else Z <= '1'; Nextstate <= S0; end if;
when S6 =>
if X = '0' then Z <= '1'; Nextstate <= S0;
else Z <= '0'; Nextstate <= S0; end if;
when others => null; -- should not occur
end case;
end process;

-- State Register
process (enable, CLK)
begin
if enable = '0' then
State <= S0;
elsif rising_edge (CLK) then
State <= Nextstate;
end if;
end process;
end Behavioral;

```

Note that in each branch of the case statement, the output **Z** and **Nextstate** are assigned values. The second process represents the state register, which is updated on the rising edge of the **CLK** signal.

Part 2 – Introduction, specification, design and testing of Moore FSM

In a Moore circuit, the outputs depend only on the present state. The most common way of schematically representing a Mealy sequential circuit is shown in Fig.3.3

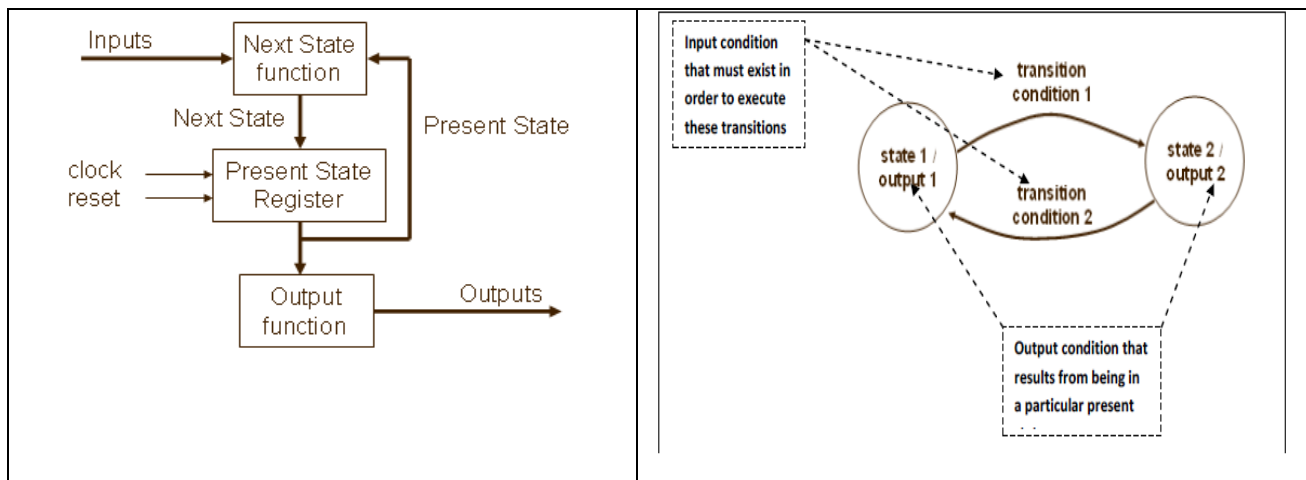


Figure 3.3 Generalized Moore Sequential circuit

In this task, you will be required to understand and complete the behavioral VHDL code for the Moore FSM. The state diagram for the FSM is given below. Since the synthesizer will automatically take care of the state encoding, therefore, the states only need to be labelled with their logical names.

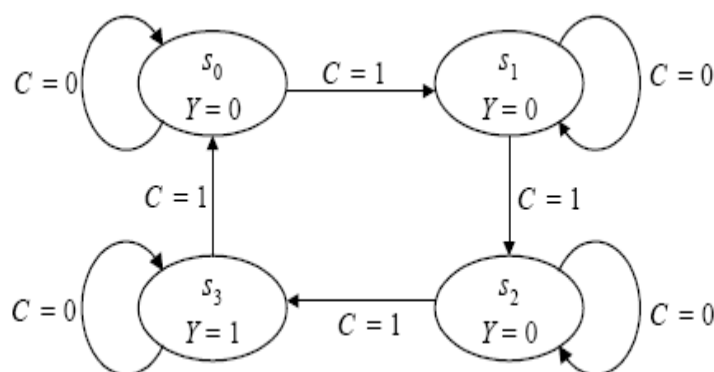


Figure 3.4 Moore Finite State Machine-an example

The behavioral VHDL code **[incomplete]** for this Moore FSM based on this state diagram is given below.

✓ *Understand and complete this code. Verify the output from the simulation*

Code:

```
ENTITY MooreFSM IS
PORT (
clock: IN STD_LOGIC;
reset: IN STD_LOGIC;
C: IN STD_LOGIC;
Y: OUT STD_LOGIC);
END MooreFSM;
```

```
ARCHITECTURE Behavioral OF MooreFSM IS
TYPE state_type IS (s0, s1, s2, s3); --user defined data type
```

```

SIGNAL state: state_type;
BEGIN
next_state_logic: PROCESS (clock, reset)
BEGIN
IF (reset = '1') THEN
state <= s0;
ELSIF (clock'EVENT AND clock = '1') THEN
CASE state IS
WHEN s0 =>
IF C = '1' THEN
state <= s1;
ELSE
state <= s0;
END IF;
WHEN s1 =>
IF C = '1' THEN
state <= s2;
ELSE
state <= s1;
END IF;
---
---
---
---
complete this code
---
---

END CASE;
END IF;
END PROCESS;
output_logic: PROCESS (state)
BEGIN
CASE state IS
WHEN s0 =>
Y <= '0';
WHEN s1 =>
Y <= '0';
---
---
complete this code
---

END CASE;
END PROCESS;
END Behavioral;

```

Practice Tasks: Transfer VHDL code into the FPGA board

1. Design and code in VHDL the converter from Example 1 but as a Moore machine. Verify its operation using Quartus simulator. The lab report should include state diagram, VHDL code, description, waveform, and synthesis report.
2. Design a 2-bit Up/Down Gray Code Counter using User-Enumerated State Encoding
 - In=0, Count Up
 - In=1, Count Down
 - this will be a Moore or mealy? Which is easier?
 - no Reset

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 4: To design the counter and sequence detector using VHDL and reproduce the sequence on the FPGA board

Objectives

The objective of this is to *design* the counter and sequence detector using VHDL

Part 1: *Design* and testing of a counter with the help of FSM.

Part 2: *Design* and testing of a sequence detector with the help of FSM.

Part 1 – Introduction, specification, design and testing of counter

Counters may be designed supporting different count directions (up or down), different count sequences (binary, grey code, etc.), and with additional inputs/outputs associated with the counter. Some of the inputs could include a count direction (UP/DOWN), a count reset (synchronous or asynchronous), counter load (LOAD), and count inhibit signals. An example output could include an indicator that signals if the count value exceeds a predetermined value.

DESIGN 1: Using the behavioral VHDL coding, create an up/down counter to increment or decrement a four-bit number. The up/down counter will have the following inputs: CLEAR, CLOCK, and UP_DOWN. CLEAR will be used to clear the outputs of the counter to 0. UP_DOWN provides the up/down capability to the counter. The counter will count to the maximum value and loop again. For example, the down counter will count from value 7 to 0 and the up counter will be opposite

Draw the FSM state diagram for the counter here

Part 2 – Introduction, specification, design and testing of Sequence Detector

Design a finite state machine with one input *X* and one output *Z*. The FSM will assert its output *Z* when it recognizes the following input bit sequence: "1011". The machine will keep checking for the proper bit sequence and does not reset to the initial state after it has recognized the string. As an example the input string *X* = "..1011011..." will cause the output to go high twice: *Z* = "..0001001..". When the machine is in the state *S3* the output will go high after the arrival of a "1" at the input. Thus the output is associated with the *transitions* as indicated on the state diagram.

You have been provided with all the necessary information during the previous lab to design and write the VHL code for this task.

Draw the FSM state diagram for the sequence detector here

Practice Tasks: Transfer VHDL code into the FPGA board

- Design a 4-bit Even-Odd Up/Down Counter using several methods of implementation. The Even-Odd Up/Down counter has a clock input, *clk*, a reset input, *rst*, count enable input, *cen*, and an up/down control input, *dir*. The counter has a 4-bit output, *count*, which outputs the current count as a 4-bit binary number. The Even-Odd counter operates as follows:
 - When *rst* = '1', the count should be reset to "0000".
 - Otherwise, if the *cen* = '1', on every clock cycle the counter should count up when *dir* = '1' and count down when *dir* = '0'.
 - The Even-Odd counter cycles between counting solely even numbers and solely odd numbers. When counting up, starting from 0, the Even-Odd counter counts up only using even numbers, 0, 2, 4, Upon reaching that last even number, 14, the counter begins counting up only using odd numbers, 1, 3, 5... .
 - If *cen* = '0' the counter should keep the present value.
- Using behavioral VHDL coding, design a Moore finite state machine that detects input test vector that contains two or more consecutive 1s. If two or more 1's are detected consecutively, the output *Z* should go high. Your design should detect overlapping sequences. The input is to be named *W*, the output is to be named *Z*, a Clock input is to be used and an active low reset signal (*Resetcn*) should asynchronously reset the machine.

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

Lab 5 (a) To design the complex counter and using VHDL and reproduce the results on the FPGA board

Lab 5 (b) To design the vending machine using VHDL and reproduce the results on the FPGA board (open ended)

Objectives

The objective of this is to *design* the complex counter and vending machine using VHDL

Part 1: *Design* and testing of a complex counter with the help of FSM.

Part 2: *Design* and testing of a Vending Machine with the help of FSM.

Part 1 – Introduction, specification, design and testing of a Complex Counter

Task 1: Design of Complex Counter

The task is to create a complex counter that can count in binary or in Gray code, depending on the value of a mode input: "A synchronous 3-bit counter has a mode control input m . When $m = 0$, the counter steps through the binary sequence 000, 001, 010, 011, 100, 101, 110, 111, and repeat. When $m = 1$, the counter advances through the Gray code sequence 000, 001, 011, 010, 110, 111, 101, 100, and repeat.

You have been provided with all the necessary information during the lab to design and write the VHL code for this task

Design Analysis

Draw the FSM state diagram for the counter here.

✓ **Complete the VHDL code given below:****Code:**

```

-----
-- Three Bit Counter
-- Data Type : std_logic_vector
-- Reset : Asynchronous
-- Active : Low
-----
Library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_signed.ALL;
entity counter_sig is
port ( UP, CLK, RESET : in std_logic;
.
.
);
end;
architecture Arch_counter_sig of counter_sig is

---
---
---
```

Part 2 Lab 5 (b) – Introduction, specification, design and testing of Vending**Machine**

In this Lab, we will look at making a vending machine. Vending machines are more complicated than you might think. We know that a vending machine must remember how much money has been inserted. This means that its outputs are a function of past inputs, so it must be a sequential circuit. Probably the best way to design this circuit is as a state machine.

Problem Statement:

A soft-drink dispensing machine is one that can dispense from a selection of pre-filled drinks. Depending on the total cash entered into the machine as well as the choice (for the drinks) selected, the machine will put out a can of each type of drink selected.

Dispensing Machine Characteristics:

1. The dispensing machine accepts Rs. 1, Rs. 2, Rs. 4 and Rs. 8 only.
2. It dispenses only three types of drinks: Fanta (costing Rs. 3), Cola (costing Rs. 4) and Sprite (costing Rs. 5).

The machine has the following 10 inputs:

50 ns Clock: You will generate it in test bench and it to your system through port mapping

Reset: Asynchronous active-high reset signal. Pressing reset returns the machine to its 'IDLE' state (initial state)

Rs.1: when Rupee 1 is entered

Rs.2: when Rupees 2 are entered

Rs.4: when Rupees 4 are entered

Rs.8: when Rupees 8 are entered

Fanta: when Fanta drink is selected

Cola: when Cola drink is selected

Sprite: when Sprite drink is selected

ENTER: Once the money is entered and drink choice has been entered, asserting this input ('1') tells the machine to put out the requested drink as well as the remaining change.

The machine has the following 3 outputs:

Total money deposited: Using any output port display the value on the wave form in the end of selection of one cold drink, showing the total money deposited (in HEX) at run-time.

Change: Another output port will be set to put out the change (in BINARY) for the user when ENTER is asserted. If no drink is selected after money has been entered, then pressing ENTER returns the entire cash back to the customer.

Dispensed item: The selected drink/s is/are put out by the machine (depending upon total cash entered and drink choice made) when ENTER is asserted

Tasks:

Draw the FSM diagram of Vending Machine controller here

You should use a FSM structure to implement this design, though it is not the only option available to you. Add sufficient comments in order to make your VHDL code more readable. Excessive use of comments also greatly reduces debug effort.

1. Write a test bench code and check the output for

a. Single drink selected.

- b. Two drinks selected.
- c. Three drinks selected.
- d. Money is returned to you by the machine when you press Enter without selecting any cold drink.

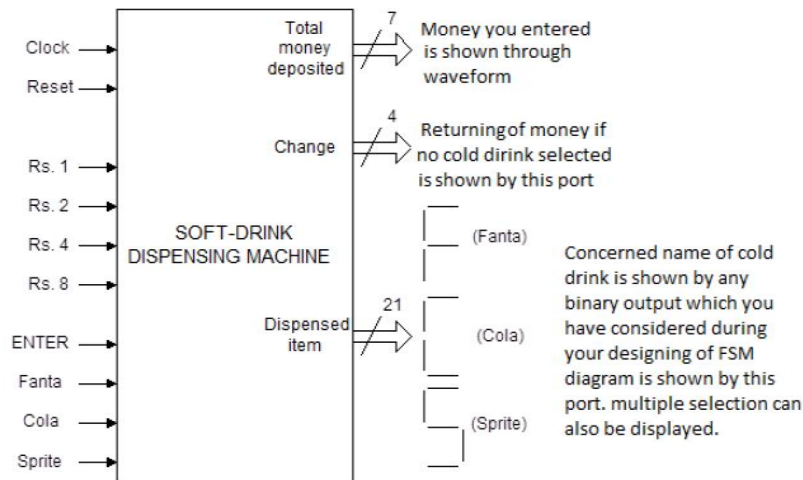


Figure 5.1 Soft Drink Dispensing Machine

Practice Tasks: Transfer VHDL code into the FPGA board

In a large system, some resources are shared by many subsystems. For example, several processors may share the same block of memory, and many peripheral devices may be connected to the same bus. An arbiter is a circuit that resolves any conflict and coordinates the access to the shared resource. This example considers an arbiter with two subsystems, as shown in figure 5.2. The subsystems communicate with the arbiter by a pair of request and grant signals, which are labeled as $r(1)$ and $g(1)$ for sub system 1, and as $r(0)$ and $g(0)$ for subsystem 0. When a subsystem needs the resources, it activates the request signal. The arbiter monitors use of the resources and the requests, and grants access to a subsystem by activating the corresponding grant signal. Once its grant signal is activated, a subsystem has permission to access the resources. After the task has been completed, the subsystem releases the resources, and deactivates the request signal. Since an arbiter's decision is based partially on the events that occurred earlier (i.e., previous request and grant status), it needs internal states to record what happened in the past. An FSM can meet this requirement. Draw the FSM for such an arbiter and write its VHDL code.

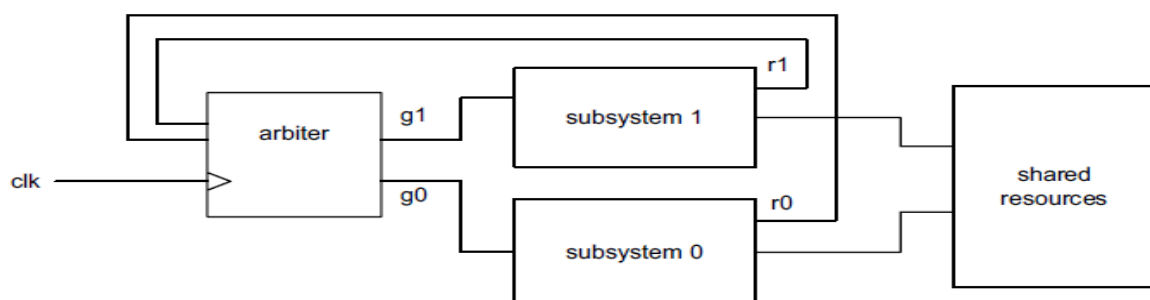


Figure 5.2 Block diagram of an arbiter

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 6: To design the array and binary multiplier using VHDL and reproduce the output on the FPGA board

Objectives

The objective of this lab is to *design*, and test design the array and binary multiplier using VHDL

Part 1: *Design* and testing of an array multiplier using VHDL

Part 2: *Design* and testing of a binary multiplier (add and shift) using VHDL

Part 1 – Introduction, specification, design and testing of an array multiplier

Task 1: Design of an array multiplier

Figure 6.1 gives an example of the traditional paper-and-pencil multiplication $P = A \times B$, where $A = 12$ and $B = 11$. We need to add two summands that are shifted versions of A to form the product $P = 132$. Part b of the figure shows the same example using four-bit binary numbers. Since each digit in B is either 1 or 0, the summands are either shifted versions of A or 0000. Figure 5.2 shows how each summand can be formed by using the Boolean AND operation of A with the appropriate bit in B .

$$\begin{array}{r} \\ \times \\ \hline \\ 1 \\ \hline 1 \end{array}$$

a) Decimal

$$\begin{array}{r} \\ \times \\ \hline \\ 1 \\ \hline 1 \end{array}$$

b) Binary

Figure 6.1

			a_3	a_2	a_1	a_0	
\times	b_3	b_2	b_1	b_0			
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
a_3b_3	a_2b_3	a_1b_3	a_0b_3				
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Figure 6.2

A four-bit circuit that implements $P = A \times B$ is illustrated in Figure 5.3. Because of its regular structure, this type of multiplier circuit is usually called an array multiplier. The shaded areas in the circuit correspond to the shaded columns in Figure 5.2. In each row of the multiplier AND gates are used to produce the summands, and full adder modules are used to generate the required sums.

A four-bit circuit that implements $P = A \times B$ is illustrated in Figure 5.3. Because of its regular structure, this type of multiplier circuit is usually called an array multiplier. The shaded areas in the circuit correspond to the shaded columns in Figure 5.2. In each row of the multiplier AND gates are used to produce the summands, and full adder modules are used to generate the required sums.

1. Generate the required VHDL file, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

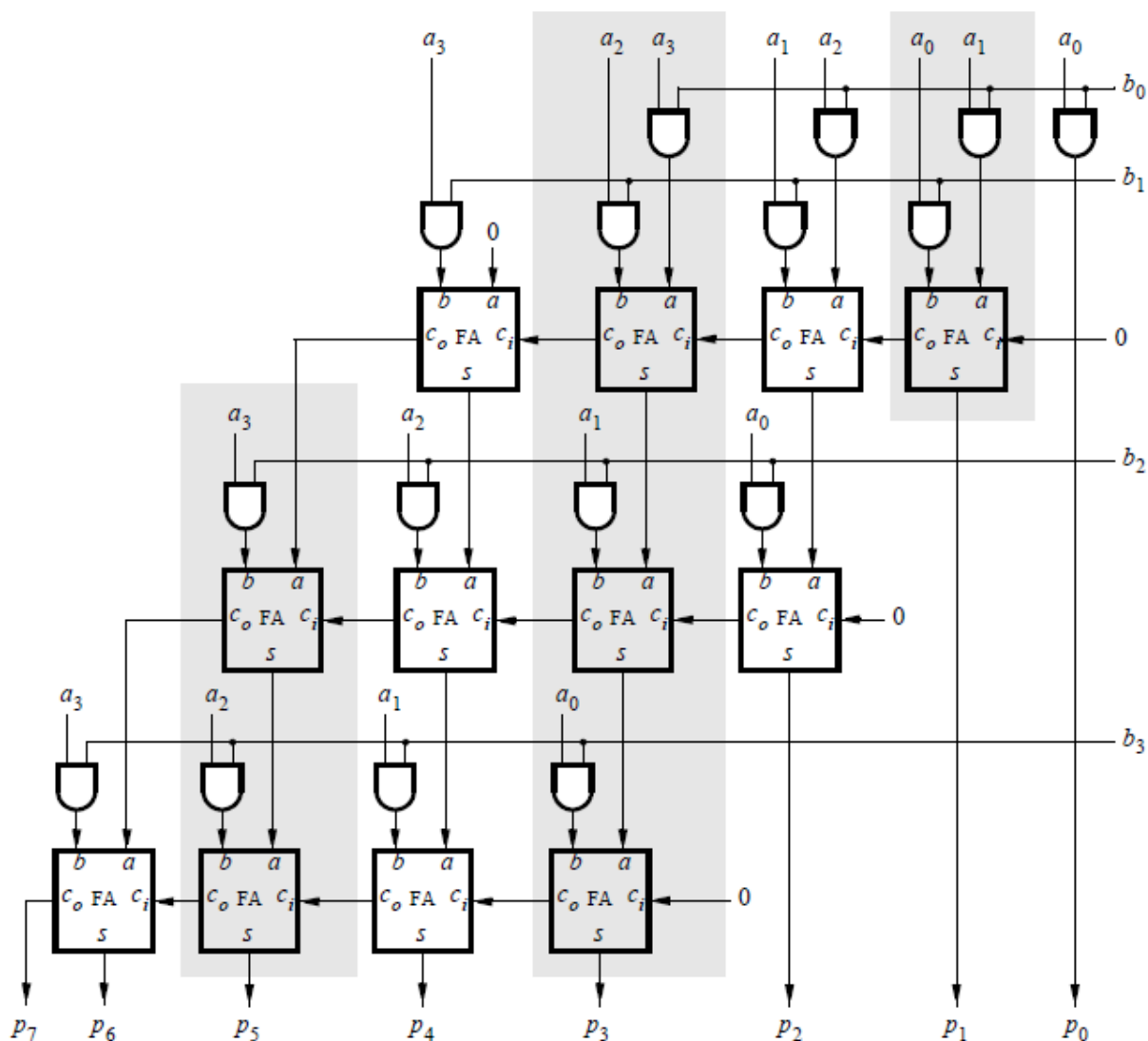


Figure 6.3

Part 2 – Introduction, specification, design and testing of binary multiplier (Shift and Add Multiplier)

Next, we will design a binary multiplier for positive binary numbers. As illustrated in last section, binary multiplication requires only shifting and adding. The following example shows how each partial product is added in as soon as it is formed. This eliminates the need for adding more than two binary numbers at a time.

$$\begin{array}{rcl}
 \text{Multiplicand} & \longrightarrow & 1101 \quad (13) \\
 \text{Multiplier} & \longrightarrow & \underline{101} \quad (11) \\
 & & 1101 \\
 \text{Partial Products} & \left\{ \begin{array}{l} \nearrow \\ \searrow \end{array} \right. & \begin{array}{r} 1101 \\ 10011 \\ \hline 0000 \\ 10011 \\ \hline 1101 \end{array} \\
 \text{Product} & \longrightarrow & \underline{10001111} \quad (143)
 \end{array}$$

The multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4-bit multiplier register, and an 8-bit register for the product. The product register serves as an accumulator to accumulate the sum of the partial products. Instead of shifting the multiplicand left each time before it is added, it is more convenient to shift the product register to the right each time. Figure 5.4 shows a block diagram for such a parallel multiplier. As indicated by the arrows on the diagram, 4 bits from the accumulator and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. The adder calculates the sum of its inputs, and when an add signal (Ad) occurs, the adder outputs are stored in the accumulator by the next rising clock edge, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry (C_4) which is generated when the multiplicand is added to the accumulator. Because the lower four bits of the product register are initially unused, we will store the multiplier in this location instead of in a separate register. As each multiplier bit is used, it is shifted out the right end of the register to make room for additional product bits. The Load signal loads the multiplier into the lower four bits of ACC and at the same time clears the upper 5 bits. The shift signal (Sh) causes the contents of the product register (including the multiplier) to be shifted one place to the right when the next rising clock edge occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal (St_1) has been received. If the current multiplier bit (M) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit is 0, the addition is skipped and only the right shift occurs. The multiplication example at the beginning of this section (13×11) is reworked below showing the location of the bits in the registers at each clock time.

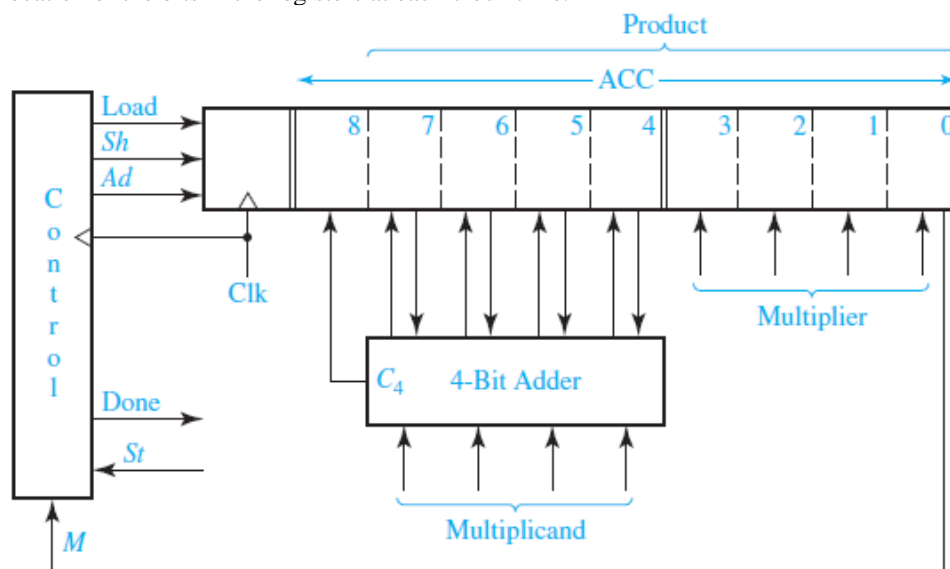


Figure 6.4 Block diagram of Binary multiplier

initial contents of product register	0 0 0 0 0 1 0 1 1 $\leftarrow M$	(11)
(add multiplicand because $M = 1$)	1 1 0 1	(13)
after addition	0 1 1 0 1 1 0 1 1	
after shift	0 0 1 1 0 1 1 0 1 $\leftarrow M$	
(add multiplicand because $M = 1$)	1 1 0 1	
after addition	1 0 0 1 1 1 1 0 1	
after shift	0 1 0 0 1 1 1 1 0 $\leftarrow M$	
(skip addition because $M = 0$)		
after shift	0 0 1 0 0 1 1 1 1 $\leftarrow M$	
(add multiplicand because $M = 1$)	1 1 0 1	
after addition	1 0 0 0 1 1 1 1 1	
after shift (final answer)	0 1 0 0 0 1 1 1 1	(143)
dividing line between product and multiplier \rightarrow		

The **control unit** must be designed to output the proper sequence of add and shift signals.

Tasks:

Draw the FSM diagram of the controller of binary multiplier here

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

Practice Tasks: Transfer VHDL code into the FPGA board

1. Consider computing the product of two 4-bit integer numbers given by $A_3A_2A_1A_0$ (multiplicand) and $B_3B_2B_1B_0$ (multiplier). Each of the ANDed terms is referred to as a partial product. The final product (the result) is formed by accumulating (summing) down each column of partial products. Any carries must be propagated from the right to the left across the columns. Since we are dealing with binary numbers, the partial products reduce to simple AND operations between the corresponding bits in the multiplier and multiplicand. The sums down each column can be implemented using one or more 1-bit binary adders. Any adder that may need to accept a carry from the right must be a full adder. If there is no possibility of a carry propagating in from the right, then a half adder can be used instead, if desired (a full adder can always be used to implement a half adder if the carry-in is tied low). The diagram below illustrates a combinational circuit for performing the 4x4 binary multiplications. The initial layer of AND gates forms the sixteen partial products that result from ANDing all combinations of the four multiplier bits with the four multiplicand bits. The column sums are formed using a combination of half and full adders. It is illustrated in the following figure. Write a structural VHDL code of this multiplier design.

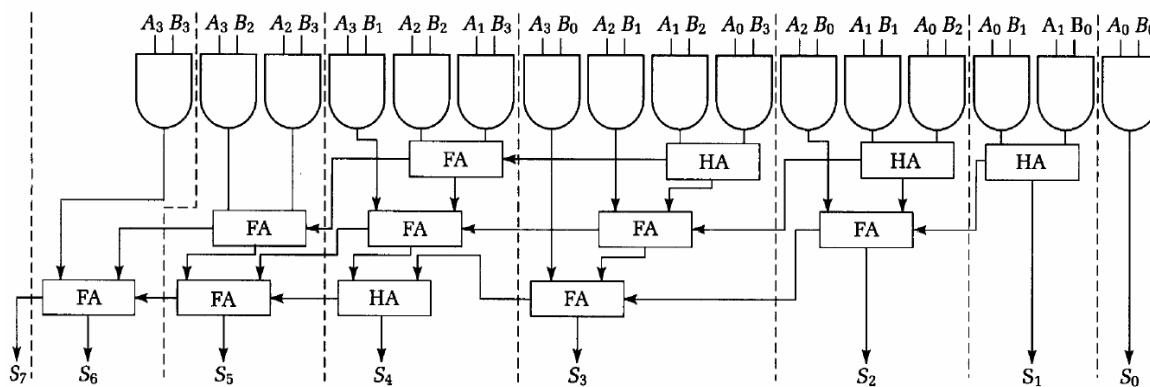


Figure 6.5 4-bit unsigned numbers multiplier

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 7: To design the signed binary multiplier using Booth's algorithm in VHDL and reproduce the results on FPGA board

Objectives

The objective of this lab is to *design*, and test signed binary number multiplier using Booth's algorithm using VHDL

Part 1 – Introduction, specification, design and testing of a signed binary number multiplier

Booth's algorithm is a multiplication algorithm which worked for two's complement numbers. It is similar to our paper-pencil method, except that it looks for the current as well as previous bit in order to decide what to do. Here are steps:

- If the current multiplier digit is 1 and earlier digit is 0 (i.e. a 10 pair) shift and sign extend the multiplicand, subtract with previous result.
- If it is a 01 pair, add to the previous result.
- If it is a 00 pair, or 11 pair, do nothing.

We start by introducing the registers that we use to implement the algorithm. The value N is the number of bits in each number being multiplied.

- SC is the sequence counter
- AC is a two bit register
- A and B are initially assigned the numbers to be multiplied together. The N bit multiplier is loaded into A. The multiplicand is sign extended to N+1 bits. In performing sign extension, the left most bits BN and BN-1 have the same value.
- Registers A and AC concatenated together are referred to as A:AC, hence A:AC is N+2 bits long.
- P is 2N bits long and will contain the final product. The upper N+1 bits are called PH and the lower N-1 bits are called PL.

We next list the steps used to perform the algorithm. In reading Booth's Algorithm it won't be clear at first why it works. To help, an example and short discussion are also presented.

Steps	A	AC	PH	PL	SC
1	11011	00	000000	0000	5
2	11101	10	000000	0000	
		111110+			

3,4			111110	0000	4
2,3,4	11110	11	111111	0000	3
2	11111	01	111111	1000	
		000010+			

3,4		000001	1000	2
2	11111	10	000000	1100
		111110+		

3,4		111110	1100	1
2,3,4	11111	11	111111	0110 0

1. INIT LOAD: Each register is loaded an initial value. The numbers to be multiplied together are loaded. One number is loaded into A, the other is sign extended to N+1 bits in length and is loaded into B. The value N is assigned to the SC register. Zeros are loaded into the P and AC registers.
2. ARITHMETIC SHIFT: The operation ashift is called an arithmetic right shift which preserves the sign bit of a two's complement number. Thus a positive number remains positive, and a negative number remains negative. Perform an ashift on P. Perform an ashift on A:AC.
3. ADD, SUBTRACT, OR PASS: If AC equals 01b then add B to PH, otherwise if AC equals 10b then subtract B from PH. To subtract, form the two's complement and then add.
4. EXIT TEST: Decrement SC, if the result is not zero then go to step 2, otherwise exit.

For this example N equals 5 and we are multiplying -5 (11011b) by +2 (00010b). The binary value 11011b is assigned to A and the binary value 00010b is sign extended to become 000010b and is assigned to B. The 2's complement of B is 111110b.

In examining the result, you should recognize that 111110110 is the two's complement representation of -10d. Now that you have seen Booth's Algorithm, we can make some comments. Booth's Algorithm is based on two principles. First, the fact that a string of zeros in A does not require any addition or subtraction but just shifting. Also, a string of ones in A is associated with a combination of subtraction (where the string begins) and addition (where the string ends).

With the previous comments in mind, you will want to be aware of more efficient versions of Booth's Algorithm. While this version uses an N+1 bit binary adder, a more efficient version makes use of only an N bit binary adder. In such an algorithm, special handling is required for cases associated with two's complement overflow. A second improvement is made by taking advantage of the fact that every iteration includes a shift operation, hence steps 2 and 3 of this version of Booth's algorithm can be combined to reduce overall execution time. A third improvement is made by eliminating the A register, reusing part of the P register, to store the multiplier.

Task1:

Draw the FSM diagram of the controller of signed binary number multiplier here

Task2:

Draw the block diagram of the system including all the components

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

Practice Tasks: Transfer VHDL code into the FPGA board

Design a faster version of the signed binary number multiplier. Write its VHDL code and compare the synthesis results for speed of both the designs (one in the lab experiment and the one you will propose).

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 8: To design the binary and signed binary divider using VHDL and reproduce output on FPGA board

Objectives

The objective of this lab is to *design* binary divider a signed binary number divider using VHDL

Part 1: Introduction, specification, design and testing of binary divider.

Part 2: Introduction, specification, design and testing of a signed binary number multiplier.

Part 1 – Introduction, specification, design and testing of a binary divider

We will consider the design of a parallel divider for positive binary numbers. As an example, we will design a circuit to divide an 8-bit dividend by a 4-bit divisor to obtain a 4-bit quotient. The following example illustrates the division process:

$$\begin{array}{r}
 \text{divisor } 1101 \overline{) 10000111} \\
 \underline{1101} \\
 0111 \\
 \underline{0000} \\
 1111 \\
 \underline{1101} \\
 0101 \\
 \underline{0000} \\
 0101 \\
 \underline{0101} \\
 0000
 \end{array}$$

(135 ÷ 13 = 10 with a remainder of 5)

quotient 1010
dividend 10000111
remainder 0101

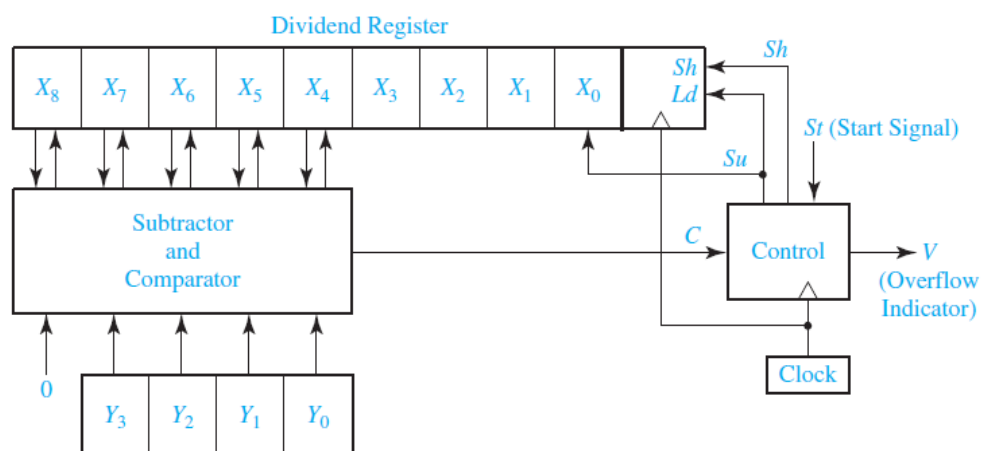
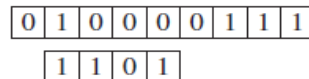
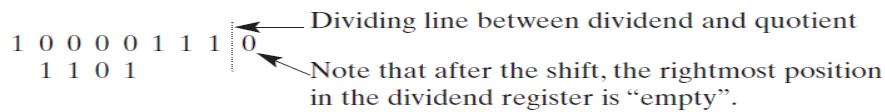


Figure 8.1 Block diagram of Binary divider

Just as binary multiplication can be carried out as a series of add and shift operations, division can be carried out by a series of subtraction and shift operations. To construct the divider, we will use a 9-bit dividend register and a 4-bit divisor register, as shown in Figure 8.1. During the division process, instead of shifting the divisor to the right before each subtraction as shown in the preceding example, we will shift the dividend to the left. Note that an extra bit is required on the left end of the dividend register so that a bit is not lost when the dividend is shifted left. Instead of using a separate register to store the quotient, we will enter the quotient bit-by-bit into the right end of the dividend register as the dividend is shifted left. Circuits for initially loading the dividend into the register will be added later. The preceding division example (135 divided by 13) is now reworked, showing the location of the bits in the registers at each clock time. Initially, the dividend and divisor are entered as follows:



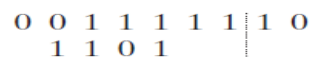
Subtraction cannot be carried out without a negative result, so we will shift before we subtract. Instead of shifting the divisor one place to the right, we will shift the dividend one place to the left:



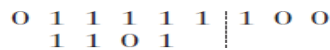
Subtraction is now carried out, and the first quotient digit of 1 is stored in the unused position of the dividend register:



Next, we shift the dividend one place to the left:



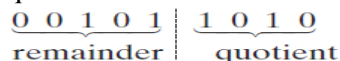
Because subtraction would yield a negative result, we shift the dividend to the left again, and the second quotient bit remains 0:



Subtraction is now carried out, and the third quotient digit of 1 is stored in the unused position of the dividend register:



A final shift is carried out and the fourth quotient bit is set to 0:



The final result agrees with that obtained in the first example. Note that in the first step the leftmost 1 in the dividend is shifted left into the leftmost position (X8) in the X register. If we did not have a place for this bit, the division operation would have failed at this step because $0000 < 1101$. However, by keeping the leftmost bit in X8, $10000 \geq 1101$, and subtraction can occur.

Task:

Draw the FSM diagram of binary divider controller circuit

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct

Part 2 – Introduction, specification, design and testing of a signed binary number divider

We now design a divider for signed (2's complement) binary numbers that divides a 32-bit dividend by a 16-bit divisor to give a 16-bit quotient. Although algorithms exist to divide the signed numbers directly, such algorithms are rather complex. So we take easy way out and complement the dividend and divisor if they are negative; when division is complete, we complement the quotient if it should be negative.

Figure 8.2 shows a block diagram for the divider. We use a 16 bit bus to load the registers. Since the dividend is 32 bits, two clocks are required to load the upper and lower halves of the dividend register, and one clock is needed to load the divisor. An extra sign flip flop is used to store the sign of the dividend. We will use a dividend register with a built-in 2's complemener. The subtractor consists of an adder and a complemener, so subtraction can be accomplished by adding 2's complement of the divisor to the dividend register. If the divisor is negative, using a separate step to complement it is unnecessary; we can simply disable the complemener and add the negative divisor instead of subtracting its complement. The control network is divided into two parts – a main control, which determines the sequence of shifts and subtracts, and a counter, which counts the number of shifts. The counter counts a signal $K=1$ when 15 shifts have occurred. Control signals are defined as follows:

LdU	Load upper half of dividend from bus
LdL	Load lower half of dividend from bus
Lds	Load sign of dividend into sign flip flop
S	Sign of dividend
CmL	complement dividend register (2's complement)
Ldd	Load divisor from bus
Su	Enable adder output onto bus (Ena) and load upper half of dividend from bus
Cm2	Enable complemener. (Cm2 equals the complement of the sign bit of the divisor, so a positive divisor is complemented and a negative divisor is not.)
Sh	Shift the dividend register left one place and increment the counter
C	Carry output from adder. If $C=1$ the divisor can be subtracted from the upper dividend
St	Start
V	Overflow

Qneg Quotient will be negative. (Qneg=1 when the sign of the dividend and divisor are different)

Procedure for carrying out the signed division is as follows:

1. Load the upper half of the dividend from the bus, and copy the sign of the dividend in to the sign flip flop.
2. Load the lower half of the dividend from the bus
3. Load the divisor from the bus
4. Complement the dividend if it is negative
5. If an overflow condition is present, go to the done state.
6. Else carry out the division by a series of shifts and subtracts
7. When division is complete, complement the quotient if necessary, and go to the done state.

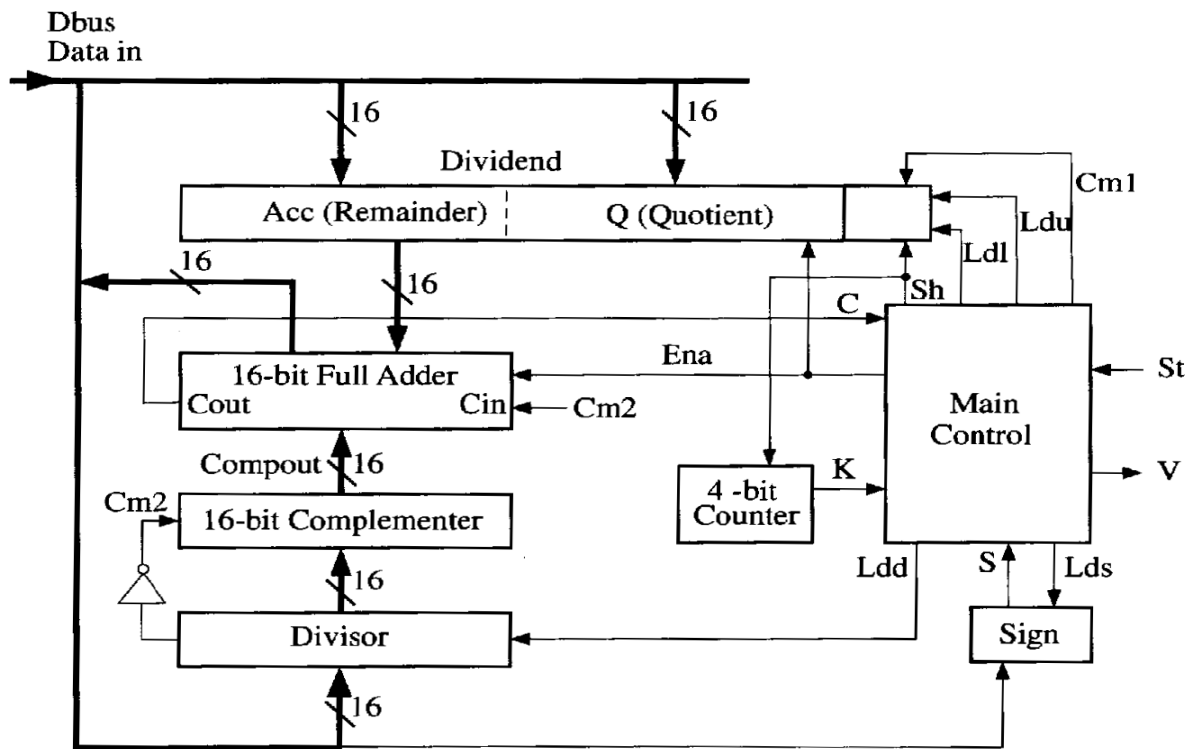


Figure 8.2 Block diagram of Signed divider

Draw the FSM diagram of the controller

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

Practice Tasks: Transfer VHDL code into the FPGA board

This exercise concerns the design of a divider for unsigned binary numbers that will divide a 16-bit dividend by an 8-bit divisor to give an 8-bit quotient. Assume that the start signal (ST=1) is 1 for exactly one clock cycle time. If the quotient would require more than 8 bits, the divider should stop immediately and output V=1 to indicate an overflow. Use a 17-bit dividend register and store the quotient in the lower 8 bits of this register. Use a 4-bit counter to count the number of shifts, together with a subtract-shift controller.

- a) Draw the block diagram of the divider.
- b) Draw the state graph for the subtract-shift controller (3 states)
- c) Write a VHDL description of the divider.

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 9: To design the traffic light controller using VHDL and reproduce the output on the FPGA board

Objectives

The objective of this lab is to *design* and test traffic light controller using VHDL.

Part 1 - Introduction, specification, design and testing of two way traffic light controller

Digital controllers are good examples of circuits that can be efficiently implemented when modelled as state machines. In the present experiment, we want to design a Traffic Light Controller (TLC) with the following characteristics and also summarized in the table 9.1

Three modes of operation: Regular, Test, and Standby.

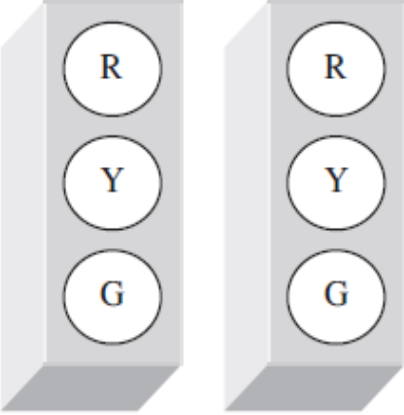
Regular Mode: Four states, each with an independent, programmable time, passed to the circuit by means of a CONSTANT.

Test mode: allows all pre-programmed times to be overwritten (by a manual switch) with a small value, such that the system can be easily tested during maintenance (1 second per state). This value should also be programmable and passed to the circuit using a CONSTANT.

Standby Mode: If set (by a sensor accusing malfunctioning, for example, or a manual switch) the system should activate the yellow lights in both directions and remain so while the standby signal is active.

Assume that a 60 Hz clock is available. However, you can also use clock divider to acquire the desired clock frequency.

Table 9.1 Specification of TLC

	Operation Mode		
	Regular	Test	Standby
	Time	Time	Time
RG	timeRG(30 s)	timeTEST(1 s)	--
RY	timeRY(5 s)	timeTEST(1 s)	--
GR	timeGR(45 s)	timeTEST(1 s)	--
YR	timeYR(5 s)	timeTEST(1 s)	--
YY	--	--	Indefinite

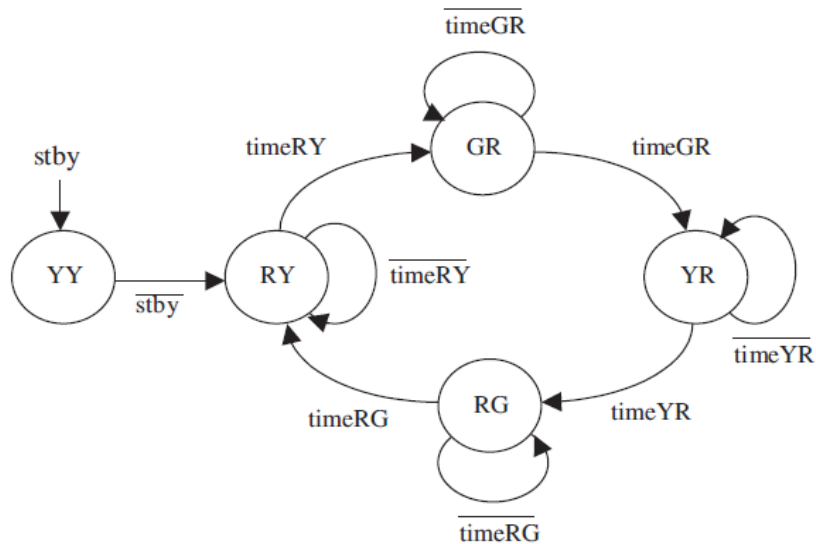


Figure 9.1 Finite State Machine Diagram of TLC

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

Practice Tasks: Transfer VHDL code into the FPGA board

1. Modify your design in part-1 for four way traffic light controller as

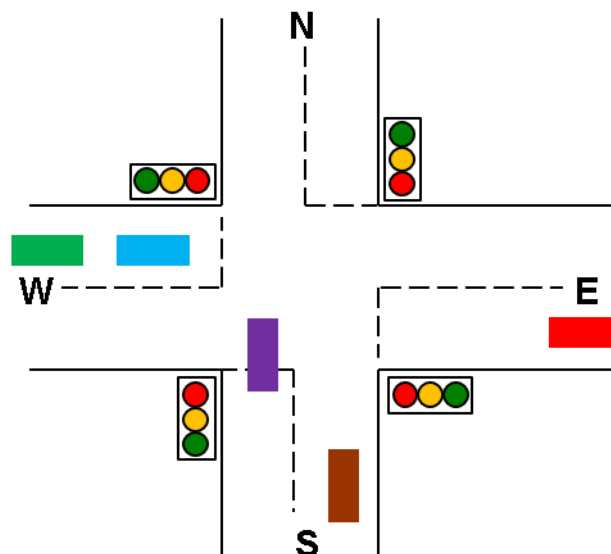


Figure 9.2 Four Way Traffic Light System

Modify your FSM and redraw it here

- a) Generate the required structural VHDL code, include it in your project, and compile the circuit.
 - b) Use functional simulation to verify that your code is correct.
2. Add crosswalk requests to your state machine. By default, the “don’t walk” signal should be on. When a crosswalk button is pressed, a walk signal should turn on the next time the parallel traffic signal turns green. (Hint: you can add another register to remember if the request buttons have been pressed during the current traffic cycle.) Present the updated state diagram and modify your VHDL design accordingly.

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 10: To design the memories (RAM and ROM) using VHDL and reproduce output on FPGA board

Objectives

The objective of this lab is to *design* and test RAM and ROM using VHDL

Part 1: Introduction, specification, design and testing of Random Access Memory (RAM)

Part 2: Introduction, specification, design and testing of Read Only Memory (ROM)

Part 1 – Introduction, specification, design and testing of RAM

Task 1 : Design of RAM [Register File]

In this task, you are to design a 16*32 RAM module. The block diagram of this module is given as.

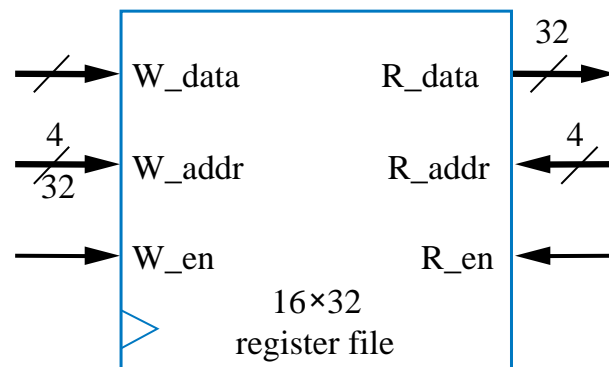


Figure 10.1 block diagram of RAM

The code for the 4*4 RAM is given below

Code:

```

-----
-- a simple 4*4 RAM module
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-----

entity SRAM is
generic( width: integer:=4;
         depth: integer:=4;
         addr: integer:=2);
port( Clock: in std_logic;
      Enable: in std_logic;

      Read: in std_logic;
      Write: in std_logic;
      Read_Addr: in std_logic_vector(addr-1 downto 0);
      Write_Addr: in std_logic_vector(addr-1 downto 0);
      Data_in: in std_logic_vector(width-1 downto 0);
      Data_out: out std_logic_vector(width-1 downto 0)
);

```

```

end SRAM;
-----
architecture behav of SRAM is
-- use array to define the bunch of internal temporary signals
type ram_type is array (0 to depth-1) of
    std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type;
begin
    -- Read Functional Section
    process(Clock, Read)
    begin
        if (Clock'event and Clock='1') then
            if Enable='1' then
                if Read='1' then
                    Data_out <= tmp_ram(conv_integer(Read_Addr));---what is purpose of this line??
                else
                    Data_out <= (Data_out'range => 'Z');
                end if;
            end if;
        end if;
    end process;

    -- Write Functional Section
    process(Clock, Write)
    begin
        if (Clock'event and Clock='1') then
            if Enable='1' then
                if Write='1' then
                    tmp_ram(conv_integer(Write_Addr)) <= Data_in;
                end if;
            end if;
        end if;
    end process;
end behav;
-----

```

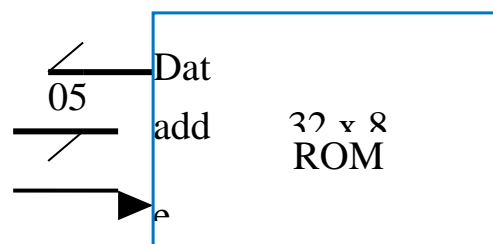
✓ *Simulate the given code for different input data and addresses*

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

Part 2 – Introduction, specification, design and testing of ROM

Task 2 : Design of ROM

In this task, you are to design a 32*8 ROM module. The block diagram of this module is given as.



ROM block symbol

Figure 10.2 ROM

Code:

```

-----
-- 32*8 ROM module
-- ROM model has predefined content for read only purpose
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ROM is
port(
    Clock   : in std_logic;
    Reset    : in std_logic;
    Enable   : in std_logic;
    Read     : in std_logic;
    Address  : in std_logic_vector(4 downto 0);
    Data_out: out std_logic_vector(7 downto 0)
);
end ROM;

-----
architecture Behav of ROM is
    type ROM_Array is array (0 to 31) of std_logic_vector(7 downto 0);
    constant Content: ROM_Array := (
        0 => "00000001",      -- Suppose ROM has
        1 => "00000010",      -- prestored value
        2 => "00000011",      -- like this table
        3 => "00000100",      --
        4 => "00000101",      --
        5 => "00000110",      --
        6 => "00000111",      --
        7 => "00001000",      --
        8 => "00001001",      --
        9 => "00001010",      --
        10 => "00001011",     --
        11 => "00001100",     --
        12 => "00001101",     --
        13 => "00001110",     --
        14 => "00001111",     --
        OTHERS => "11111111"  --
    );
begin
    process(Clock, Reset, Read, Address)
    begin
        if( Reset = '1' ) then
            Data_out <= "ZZZZZZZZ";
        elsif( Clock'event and Clock = '1' ) then
            if Enable = '1' then
                if( Read = '1' ) then
                    Data_out <= Content(conv_integer(Address));
                else
                    Data_out <= "ZZZZZZZZ";
                end if;
            end if;
        end if;
    end process;
end Behav;
-----

```

✓ **Simulate the given code for different input data and addresses**

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

Practice Tasks: Transfer VHDL code into the FPGA board

Write a VHDL code for a memory controller shown in the form of FSM in figure 10.2. The controller is between a processor and a memory chip, interpreting commands from the processor and then generating a control sequence accordingly. The commands, *mem*, *rw* and *burst*, from the processor constitute the input signals of the FSM. The *mem* signal is asserted to high when a memory access is required. The *rw* signal indicates the type of memory access, and its value can be either '1' or '0', for memory read and memory write respectively. The *burst* signal is for a special mode of memory read operation. If it is asserted, for consecutive read operations will be performed. The memory chip has two control signals, *oe* (for input enable) and *we* (for write enable), which need to be asserted during the memory read and memory write respectively. The two output signals of the FSM, *oe* and *we*, are connected to the memory chip's control signals. For comparison purpose, we also add an artificial Mealy output signal, *we_me*, to the state diagram.

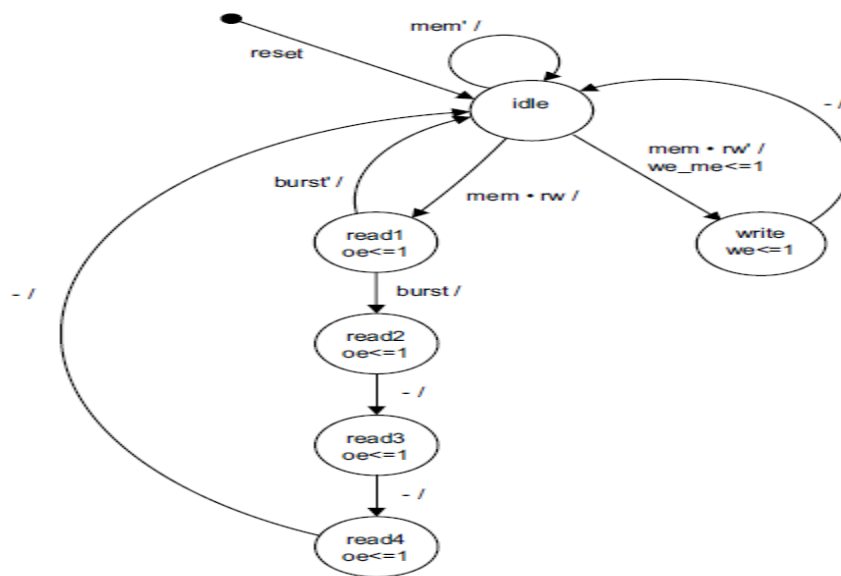


Figure 10.3 state diagram of a memory controller

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 11: To design the keypad scanner using VHDL and reproduce the output on the FPGA board

Objectives

The objective of this lab is to *design* a scanner for a telephone keypad using VHDL

Part-1 Introduction, specification, design and testing of a Complex Counter

In this lab, you are going to design a scanner for a telephone keypad using VHDL. Figure 11.1 shows a block diagram for the system.

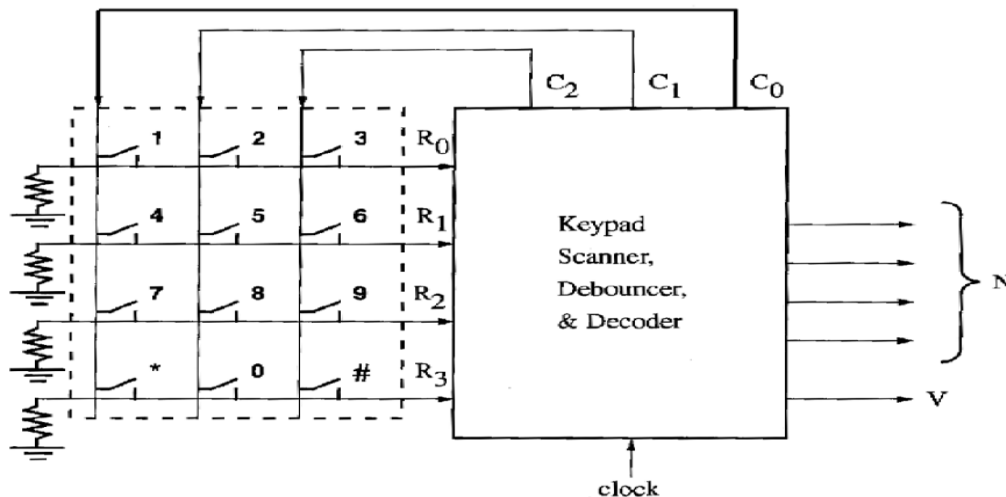


Figure 11.1 Block diagram of keypad scanner

The keypad is wired in matrix form with a switch at the intersection of each row and column. Pressing a key establishes a connection between a row and column. The purpose of the scanner is to determine which key has been pressed and output a binary number $N = N_3 N_2 N_1 N_0$, which corresponds to the key number. For example, pressing key 5 will output 0101, pressing the * key will output 1010, and pressing the # key will output 1011. When a valid key has been detected, the scanner should output a signal V for one clock time. We will assume that only one key is pressed at time.

We will use the following procedure to scan the keyboard: First apply logic 1s to columns C₀, C₁, and C₂ and wait. If any key is pressed, a 1 will appear on R₀, R₁, R₂, or R₃. Then apply a 1 to column C₀ only. If any of the R_i's is 1, a valid key is detected, so set V=1 and output the corresponding N. If no key is detected in the first column, apply a 1 to C₁ and repeat. If no key is detected in the second column, repeat for C₂. When a valid key is detected, apply 1s to C₀, C₁, and C₂ and wait until no key is pressed. This last step is necessary so that only valid signal is generated each time key is pressed.

In the process of scanning the keyboard to determine which key is pressed, the scanner must take contact bounce into account. When a mechanical switch is closed or opened, the switch contact will bounce, causing noise in the switch output, as shown in the figure 11.2. The contact may bounce for several ms before it settles down to its final position. After a switch closure has been detected, we must wait for the bounce to settle before reading the key. The signal that indicates a key has been pressed also should be synchronized with the clock, since it will be used as an input to a synchronous sequential network.

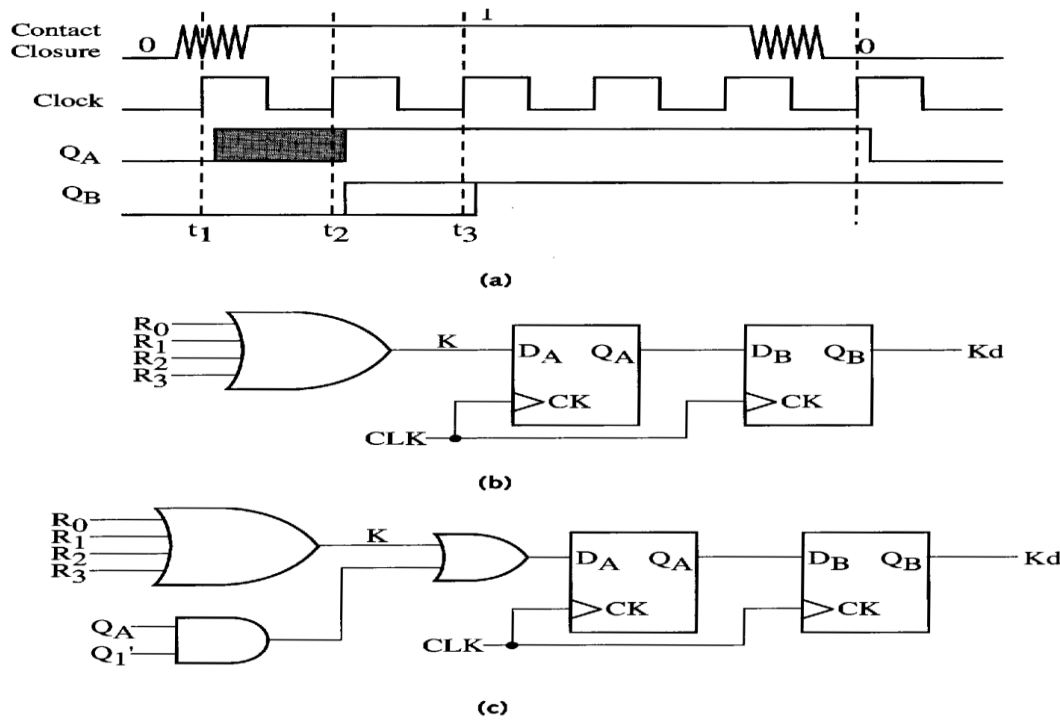


Figure 11.2 Debouncing and synchronizer circuit

Figure 11.2 (b & c) shows a proposed debouncing and synchronizing circuit. The clock period must be greater than the bounce time. If $C_0 = C_1 = C_2 = 1$, when any key is pressed, K will become 1 after the bounce is settled. If the rising edge of the clock occurs during the bounce, either a 0 or 1 will be clocked into the flip flop at t_1 . If a 0 was clocked in, a 1 will be clocked in at the next active clock edge (t_2). So it appears that Q_A will be debounced and synchronized version of K. However, a possibility of failure exists if K changes very close to the clock edge such that the setup or hold time is violated. In this case the flip flop output Q_A may oscillate or otherwise malfunction. Although this situation will occur very infrequently, it is best to guard against it by adding a second flip flop. We will choose the clock period so that any oscillation at the output of Q_A will have died out before the next active edge of the clock so that the input D_B will always be stable at the active clock edge. The debounced signal K_d , will always be clean and synchronized with the clock, although it may be delayed up to two clock cycles after the key is pressed.

The keypad scanner design is divided in to three modules as shown in figure 11.3. The debounce module generates a signal K when a key has been pressed and a signal K_d after it has been debounced. The keyscan module generates the column signals to scan the keyboard. When a valid key is detected, the decoder determines the key number from the row and column numbers.

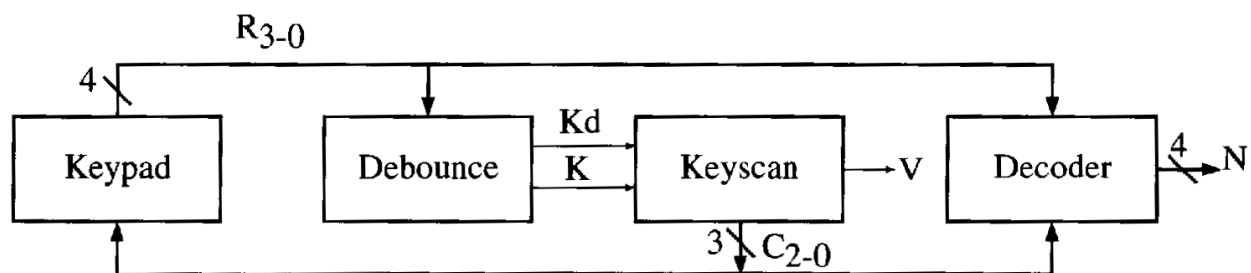


Figure 11.3 Scanner modules

Part- 2 FSM of controller and decoder design of keypad scanner**Task 1:**

Draw the FSM diagram of the keypad controller

The decoder determines the key number from the row and column numbers using the truth table given in Table 11.1. The truth has one row for each of the 12 keys. The remaining rows have don't care outputs since it has been assumed that only one key is pressed at a time. Since the decoder is a combinational network, its output will change as the keypad is scanned.

Table 11.1 Truth table of decoder

R_3	R_2	R_1	R_0	C_0	C_1	C_2	N_3	N_2	N_1	N_0
0	0	0	1	1	0	0	0	0	0	1
0	0	0	1	0	1	0	0	0	1	0
0	0	0	1	0	0	1	0	0	1	1
0	0	1	0	1	0	0	0	1	0	0
0	0	1	0	0	1	0	0	1	0	1
0	0	1	0	0	0	1	0	1	1	0
0	1	0	0	1	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0
0	1	0	0	0	0	1	1	0	0	1
1	0	0	0	1	0	0	1	0	1	0 (*)
1	0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	0	1	1	0	1	1 (#)

Task: 2

Find the logic equations of the decoder

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

LAB # 12: To design the finite impulse response (FIR) filter using VHDL and reproduce output on vector waveform

Objective

The objective of this lab is to *design* and test different architectures of Finite Impulse Response (FIR) filter

Part 1: Introduction, specification, design and testing of direct form FIR filter

Part 2: Introduction, specification, design and testing of time multiplexed architecture of FIR filter.

Part 1 – Introduction, specification, design and testing of direct form FIR filter

Introduction

In this part, you will design a 16-tap FIR filter in VHDL and will simulate it in Quartus. The filter should be implemented once in direct form and then pipelining should be applied. The structure of a direct form FIR filter is given in Figure 12.1.

The FIR Filter

The FIR filter is a digital filter with a finite duration impulse response. These types of filters have applications in digital signal processing and the communication systems, where a linear phase is required in the pass band. The FIR filter algorithm is actually a convolution of the input signal with a set of pre-defined coefficients. Mathematically, the FIR filtering process can be described as

$$y[k] = \sum_{m=0}^{n-1} h[m]x[k-m]$$

where n is the number of taps. The data flow diagram of a digital FIR filter consists of a series of multipliers, adders and delay elements to store the w-bit input words as they transverse the filter. A shift register, a multiplier and an adder are used for each tap in the filter. For any multiplier in a certain tap, one input is the delayed input words and the other input is programmed with the value of the associated coefficient.

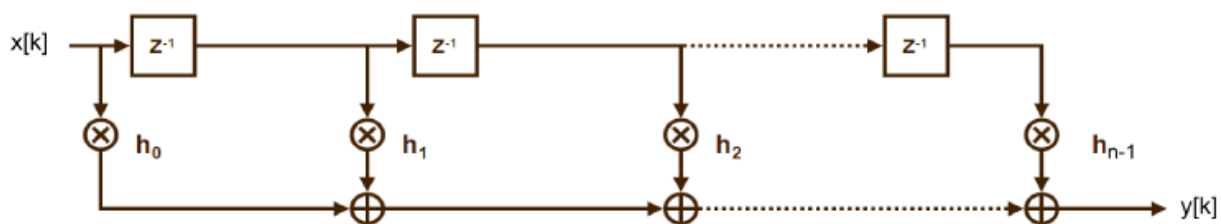


Figure 12.1 Structure of n-tap FIR filter

Task 1 Inputs of the 16-tap filter are represented by 8 bits, also 7 bits are assigned for each of the filter coefficients. What are the required word lengths for the adders and multipliers in every tap to achieve a full precision processed output? Consider the direct form structure for the filter as in Figure 12.1.

Task 2 Implement a direct form FIR filter in VHDL using the coefficients given below $H = [-1, 0, 1, -3, -9, -2, 30, 63, 63, 30, -2, -9, -3, 1, 0, -1]$

Assign 7 bits for every filter coefficients and assume an 8-bit input sequence. The impulse and step responses of such a filter are shown in Figure 12.2.

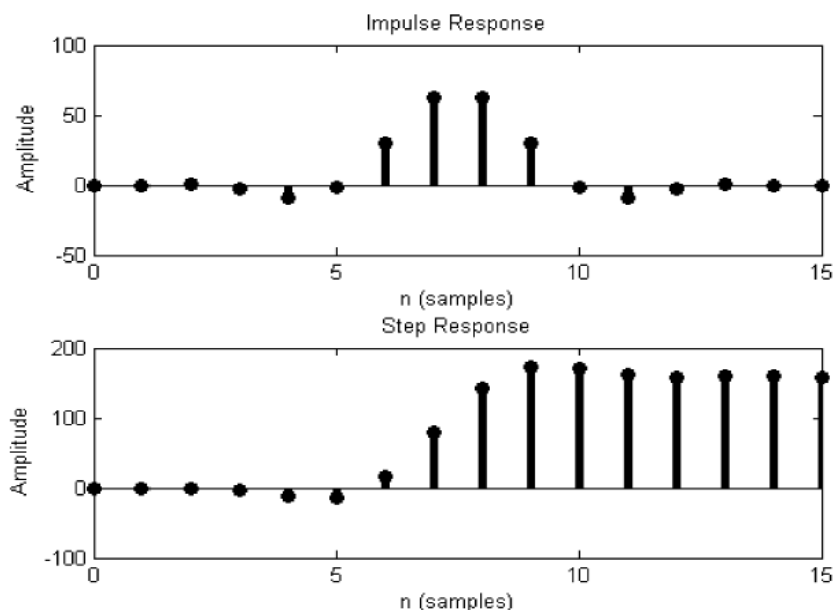
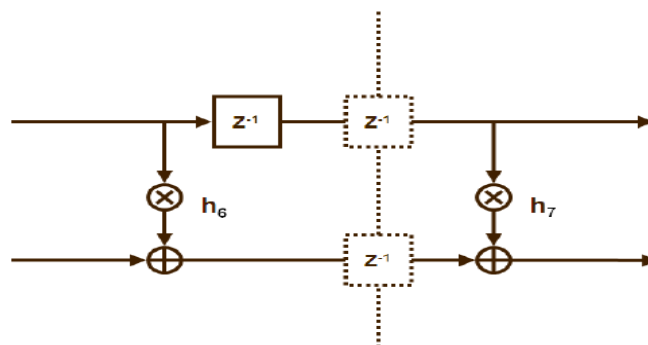


Figure 12.2 Impulse and Step response of FIR filter

The filter output must be in full precision, so you must consider the proper word length for the different taps of the filter.

Task 3

Now implement the filter in a pipelined structure in VHDL. Add a single pipeline stage between taps 7 and 8 of the filter as shown in Figure 12.3. Verify its correct functionality using Quartus likewise as the previous task. Synthesize both structures of the filter separately using the Quartus. What are the maximum possible clock frequencies for each structure? What do you gain and what do you lose from applying a pipeline stage?

Figure 12.3 A Pipeline stage between 7th and 8th Taps of the filter

Part 2 – Introduction, specification, design and testing of time multiplexed architecture of FIR filter

Figure 12.4 shows a hardware-mapped implementation of an FIR filter. The basic operation of this algorithm is shaded in grey, that is, a multiplication, followed by an addition and a register. The output is simply a superposition of partial products.

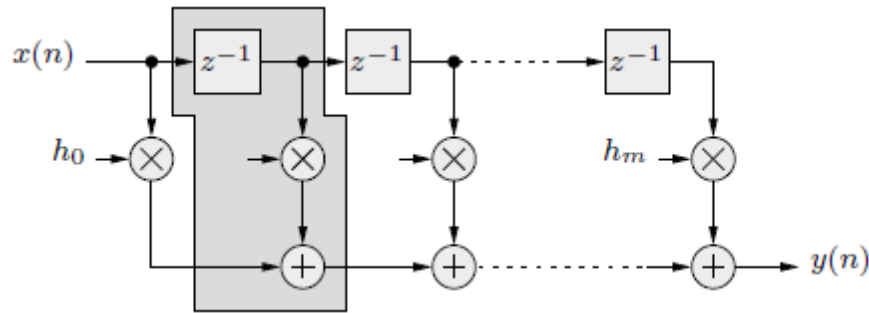


Figure 12.4 Basic architecture of an FIR filter

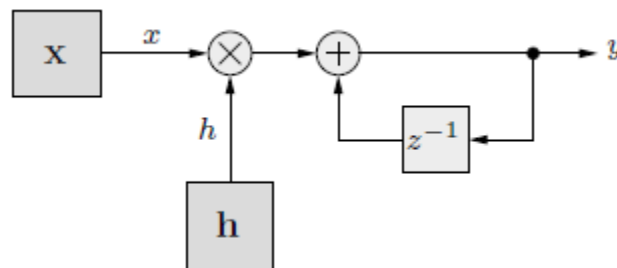


Figure 12.5 Multiply and Accumulate unit (MAC) in time multiplexed FIR filter

A time-multiplexed version is depicted in Figure 12.5. It basically consists of a multiply-accumulate (MAC) unit that iteratively calculates an output sample. Here, the input samples $\mathbf{x} = x(k), x(k-1), \dots, x(k-m)$ are held in a memory and a specific sample x has to be selected to be multiplied with the respective coefficient h . After an iteration is completed and an output sample is created, the accumulate buffer is cleared and the next iteration can begin. Now, the input memory is updated with a new sample, that is, $\mathbf{x} = x(k+1), x(k), \dots, x(k-m+1)$, and the accumulation starts over again.

Following are the area and speed properties for the filter in two different forms.

1. Hardware-mapped implementation

- One output sample is calculated in one clock cycle
- Utilizes $(m+1)$ fixed multipliers, m adders, and m registers
- Throughput fixed, size depends on m
- Each pipeline stage can be optimized separately

2. Time-multiplexed implementation

- One output sample is calculated in $(m+1)$ clock cycles
- Utilizes 1 flexible multiplier, 1 adder, and 1 register
- Throughput depends on m , size fixed
- Has to be designed for maximum wordlength w_{dyn}

1. Generate the required structural VHDL code, include it in your project, and compile the circuit.
2. Use functional simulation to verify that your code is correct.

Manchester code

Manchester code is a coding scheme used to represent a bit in a data stream. A '0' value of a bit is represented as a 0-to-1 transition, in which the lead half is '0' and the remaining half is '1'. Similarly, a '1' value of a bit is represented as a 1-to-0 transition, in which the lead half is '1' and the remaining half is '0'. A sample data stream in Manchester cod is shown in Figure 12.6. The Manchester code is frequently used in a serial

communication line. Since there is a transition in each bit, the receiving system can use the transitions to recover the clock information.

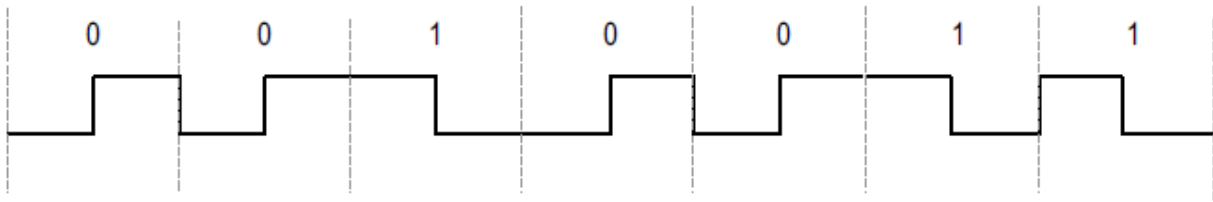


Figure 12.6 Sample waveform of Manchester encoding

The Manchester encoder transforms a regular data stream in to a Manchester-coded data stream. Because an encoded bit includes a sequence of “01” or “10”, two clock cycles are needed. Thus, the maximal data rate is only half of the clock rate. There are two input signals. The d signal is the input data stream, and the v signal indicates whether the d signal is valid (i.e., whether there is data to transmit). The d signal should be converted to Manchester code if the v signal is asserted. The output remains ‘0’ otherwise. The state diagram is shown in figure 12.7. While v is asserted, the FSM starts the encoding process. If d is ‘0’, it travels through $s0a$ and $s0b$ states. If d is ‘1’, the FSM travels through the $s1a$ and $s1b$ states. Once the FSM reached the $s1b$ or $s0b$ state and continuously encodes the next input data. The Moore output is used because we have to generate two equal intervals for each bit.

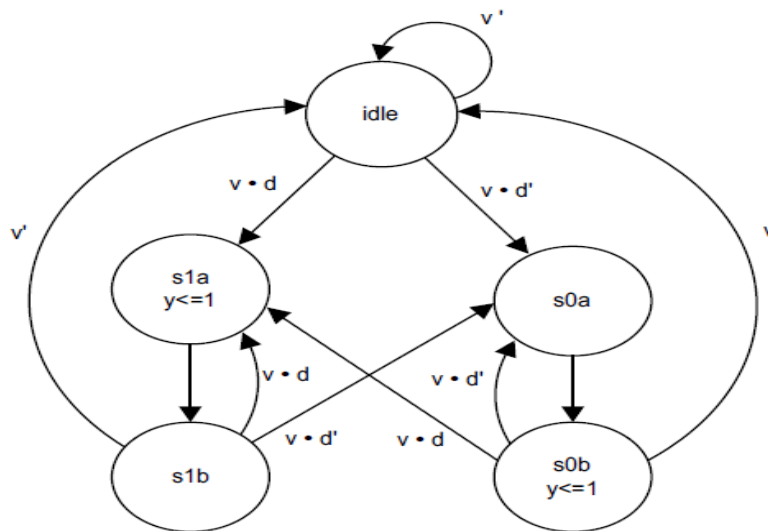


Figure 12.7 FSM diagram of a Manchester encoder

Rubric for lab assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed all the tasks and showed the results without any help of the instructor.	4	
Good	The student completed all the tasks and showed the results with minimal help of the instructor.	3	
Average	The student partially completed the task and showed results.	2	
Worst	The student did not complete the task.	1	

Instructor Signature: _____ **Date:** _____

Pin Assignment of Altera FPGA DE2 board

Pin Assignment SWITCHES

<i>Signal Name</i>	<i>FPGA Pin No.</i>	<i>Description</i>	<i>I/O Standard</i>
SW[0]	PIN_AB28	Slide Switch[0]	Depending on JP7
SW[1]	PIN_AC28	Slide Switch[1]	Depending on JP7
SW[2]	PIN_AC27	Slide Switch[2]	Depending on JP7
SW[3]	PIN_AD27	Slide Switch[3]	Depending on JP7
SW[4]	PIN_AB27	Slide Switch[4]	Depending on JP7
SW[5]	PIN_AC26	Slide Switch[5]	Depending on JP7
SW[6]	PIN_AD26	Slide Switch[6]	Depending on JP7
SW[7]	PIN_AB26	Slide Switch[7]	Depending on JP7
SW[8]	PIN_AC25	Slide Switch[8]	Depending on JP7
SW[9]	PIN_AB25	Slide Switch[9]	Depending on JP7
SW[10]	PIN_AC24	Slide Switch[10]	Depending on JP7
SW[11]	PIN_AB24	Slide Switch[11]	Depending on JP7
SW[12]	PIN_AB23	Slide Switch[12]	Depending on JP7
SW[13]	PIN_AA24	Slide Switch[13]	Depending on JP7
SW[14]	PIN_AA23	Slide Switch[14]	Depending on JP7
SW[15]	PIN_AA22	Slide Switch[15]	Depending on JP7
SW[16]	PIN_Y24	Slide Switch[16]	Depending on JP7
SW[17]	PIN_Y23	Slide Switch[17]	Depending on JP7

Pin Assignment LEDs

<i>Signal Name</i>	<i>FPGA Pin No.</i>	<i>Description</i>	<i>I/O Standard</i>
LEDR[0]	PIN_G19	LED Red[0]	2.5V
LEDR[1]	PIN_F19	LED Red[1]	2.5V
LEDR[2]	PIN_E19	LED Red[2]	2.5V
LEDR[3]	PIN_F21	LED Red[3]	2.5V
LEDR[4]	PIN_F18	LED Red[4]	2.5V
LEDR[5]	PIN_E18	LED Red[5]	2.5V
LEDR[6]	PIN_J19	LED Red[6]	2.5V
LEDR[7]	PIN_H19	LED Red[7]	2.5V
LEDR[8]	PIN_J17	LED Red[8]	2.5V
LEDR[9]	PIN_G17	LED Red[9]	2.5V
LEDR[10]	PIN_J15	LED Red[10]	2.5V
LEDR[11]	PIN_H16	LED Red[11]	2.5V
LEDR[12]	PIN_J16	LED Red[12]	2.5V
LEDR[13]	PIN_H17	LED Red[13]	2.5V
LEDR[14]	PIN_F15	LED Red[14]	2.5V

Pin Assignment Clocks

TABLE 3-1: Pin Assignment of Clock Signals

Signal Name	FPGA Pin No.	Description	I/O Standard
CLOCK_50	PIN_Y2	50 MHz clock input	3.3V
CLOCK2_50	PIN_AG14	50 MHz clock input	3.3V
CLOCK3_50	PIN_AG15	50 MHz clock input	Depending on JP6
SMA_CLKOUT	PIN_AE23	External (SMA) clock output	Depending on JP6
SMA_CLKIN	PIN_AH14	External (SMA) clock input	3.3V

Pin Assignment of Altera FPGA DE1 SOC board

Pin Assignment SWITCHES

TABLE 3-2: Pin Assignment of Switches

Signal Name	FPGA Pin No.	Description	I/O Standard
SW[0]	PIN_AB12	Slide Switch[0]	3.3V
SW[1]	PIN_AC12	Slide Switch[1]	3.3V
SW[2]	PIN_AF9	Slide Switch[2]	3.3V
SW[3]	PIN_AF10	Slide Switch[3]	3.3V
SW[4]	PIN_AD11	Slide Switch[4]	3.3V
SW[5]	PIN_AD12	Slide Switch[5]	3.3V
SW[6]	PIN_AE11	Slide Switch[6]	3.3V
SW[7]	PIN_AC9	Slide Switch[7]	3.3V
SW[8]	PIN_AD10	Slide Switch[8]	3.3V
SW[9]	PIN_AE12	Slide Switch[9]	3.3V

Pin Assignment LEDs

TABLE 3-3: Pin Assignment of LEDs

Signal Name	FPGA Pin No.	Description	I/O Standard
LEDR[0]	PIN_V16	LED [0]	3.3V
LEDR[1]	PIN_W16	LED [1]	3.3V
LEDR[2]	PIN_V17	LED [2]	3.3V
LEDR[3]	PIN_V18	LED [3]	3.3V
LEDR[4]	PIN_W17	LED [4]	3.3V
LEDR[5]	PIN_W19	LED [5]	3.3V
LEDR[6]	PIN_Y19	LED [6]	3.3V
LEDR[7]	PIN_W20	LED [7]	3.3V
LEDR[8]	PIN_W21	LED [8]	3.3V
LEDR[9]	PIN_Y21	LED [9]	3.3V

Pin Assignment CLOCKS

TABLE 3-4: Pin Assignment of Clocks

Signal Name	FPGA Pin No.	Description	I/O Standard
CLOCK_50	PIN_AF14	50 MHz clock input	3.3V
CLOCK2_50	PIN_AA16	50 MHz clock input	3.3V
CLOCK3_50	PIN_Y26	50 MHz clock input	3.3V
CLOCK4_50	PIN_K14	50 MHz clock input	3.3V
HPS_CLOCK1_25	PIN_D25	25 MHz clock input	3.3V
HPS_CLOCK2_25	PIN_F25	25 MHz clock input	3.3V